

# 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<sup>†</sup> Joshua Garcia Jiaqi Ma<sup>†</sup> Qi Alfred Chen

University of California, Irvine <sup>†</sup>University of California, Los Angeles

{ziwenw8, junjies1, jzchuang, joshua.garcia, alfchen}@uci.edu

<sup>†</sup>{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 [31–33]. 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  $3.5 \times$  average slow-down or even failure in their discoveries. We also manually verified that 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> [34]. We also discuss root causes and potential fixes. We also release our code at our website [34].

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. Specifically, we (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 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 adopts a 3-layer design: *route planning*, *behavioral planning (BP)*, and *local planning* [29, 30, 35–41]. 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 [42–44], but so far they are all experimental (e.g., designed and evaluated only for simulated racing game setups [45–49]) and generally lack the necessary capability and support for real-world driving (e.g., limited to only cruising without handling intersections, cross-walks, pedestrians [49–52], and no consideration of traffic rules [52–57]). Such a learning-based approach is also generally known to suffer from difficulties with debugging and interpreting [58], and enforcing safety rules/measures [59], 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, 60, 61]. 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 [62, 63]. In fact, such overly-conservative behaviors have *already been observed in real-world production AD settings*, causing troubles for AD service and surrounding traffic flows. For example, there are videos 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 of such a problem is in Fig. 1. Since there is no official report on the root cause, we cannot be certain that this is caused by planning flaws/bugs; however, we suspect so since the environmental perception results are all correct according to the in-vehicle display [17]. Furthermore, a few AD companies (e.g., Waymo, Volvo, Uber) have reported that



Figure 1: Real-world overly-conservative driving behavior observed for a Waymo AD vehicle, which got stuck in the middle of the road and force other vehicles to borrow the reverse lane [17].

their AD vehicles have been “bullied” by human drivers [18–20]. For example, Waymo AD vehicle found it difficult to pull away from stop signs since they were too timid and were taken advantage of by passing human drivers [18].

**Problem seriousness and incentives.** With the growing deployment of commercial AD services without safety drivers (e.g., by Waymo, Nuro, and Baidu [64–66]), such DoS problem 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). Also, from the transportation system’s perspective, such attacks can damage traffic mobility and city functions (e.g., by causing traffic jams), and also waste fuel and an individual’s time [67]. Similar to DoS attacks on the Internet, such attacks may be politically or financially incentivized [68].

**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

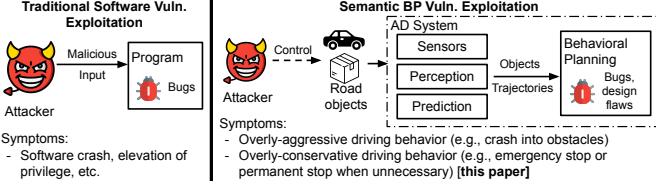


Figure 2: Illustration of the domain-specific threat model and exploitation symptoms of the semantic BP DoS vulnerabilities targeted in this paper, along with their distinctions to those of traditional software vulnerabilities.

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 perceptual 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

```

1 # Pre-defined lateral safety buffer, in meter
2 obstacle_lat_buffer ← 0.4f
3 ADC_width ← 2.11f #Default AD veh width, in meter
4 #Initialize left/right boundaries of drivable
   ← space
5 {left_bound, right_bound} ← {left_lane_boundary,
   → right_lane_boundary}
6 # Iterate over static obstacles at this
   → longitudinal position
7 for (each obs in obstacles_list)
8   if ((obs.min_l+obs.max_l)/2<0): # Left-side
      → obstacles
9     left_bound ← max(left_bound, obs.max_l +
   → obstacle_lat_buffer)
10 else: # Right-side obstacles
11   right_bound ← min(right_bound, obs.min_l -
   → obstacle_lat_buffer)
12 # Check whether exists drivable space laterally
13 if (right_bound - left_bound < ADC_width):
14   path_blocked ← true #Conclude: lane is blocked

```

Figure 3: Simplified pseudo code for a semantic DoS vulnerability PlanFuzz discovered from BP in Apollo, an industry-grade AD system [29].

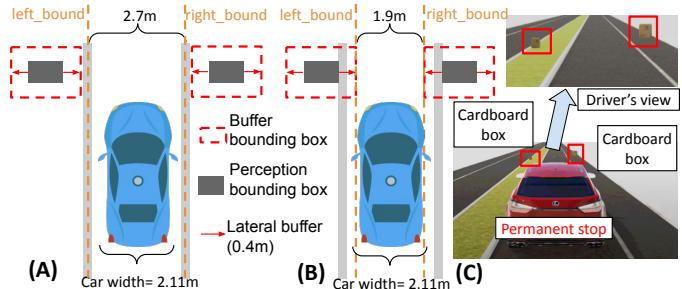


Figure 4: Illustration of (A) BP decision logic in Fig. 3; (B) its semantic BP DoS vulnerability; (C) potential real-world exploitation of it from our end-to-end simulation case study (§VI).

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 [69], and thus static obstacles out of the boundaries of such lanes can in the extreme case reduce the lateral drivable space between `left_bound` and `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. [70–73]. 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 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 *too conservative* to the extent that it *directly* violates common driving norms. For example, the requirement that a vehicle “should never block the normal and reasonable movement of traffic” [74], 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 [75].

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.

**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.

## 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, 76–80]. 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:

$$IsSafeToDrive \rightarrow \neg Stop \quad (1)$$

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:

**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, 76–78] 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 *concretize* the predicate *IsSafeToDrive* in Eq. 1 into expressions which can be directly computed from planning inputs.

**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 [81], 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 [82–86]. In the general CPS testing domain, prior works have used quantitative feedback such as the robustness metrics to guide testing [28, 79], 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, 87–89]. 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 [90].

**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

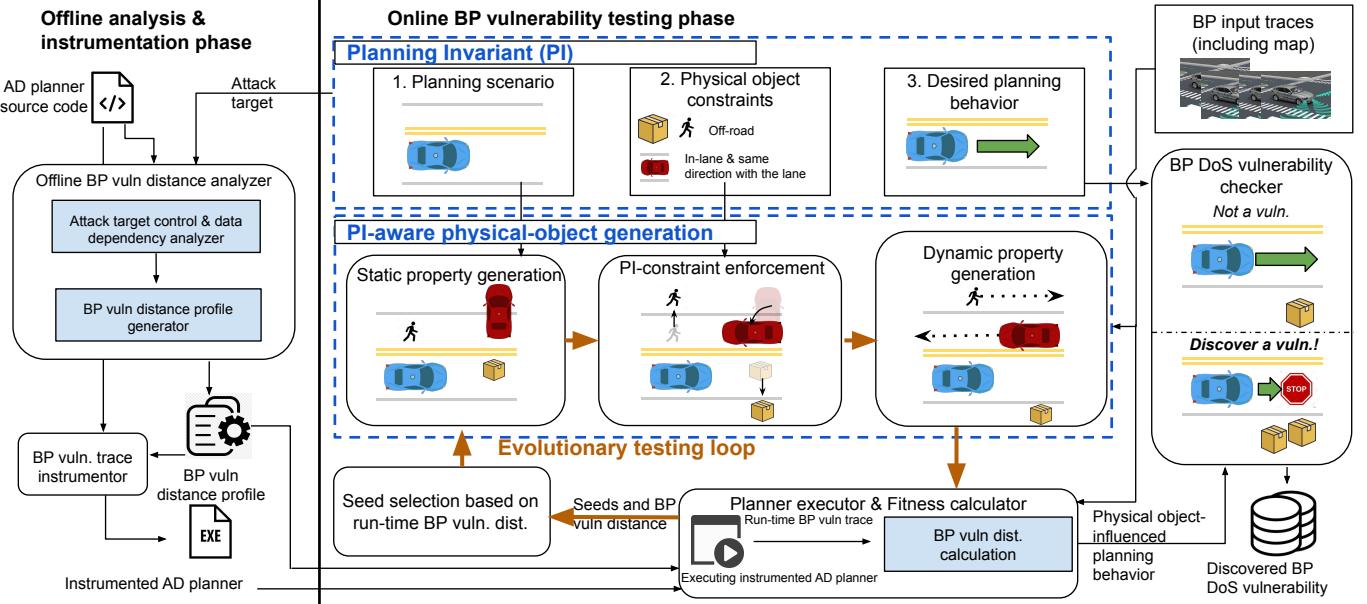


Figure 5: Overview of PlanFuzz, our automated approach for systematically discovering BP DoS vulnerabilities in AD systems.

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 class) 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

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).

Physical object type	PI constraints for physical objects
Static obstacle (cardboard boxes, parked bikes, etc.)	PI-C1. <i>StaticOffRoad(x)</i>
Vehicle	PI-C2. <i>IsFollowingVehicle(x)</i> PI-C3. <i>IrrelevantVehicle(x)</i>
Pedestrian	PI-C4. <i>StaticOffRoad(x)</i> PI-C5. <i>DynamicOffRoad(x)</i>

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. 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, 91], 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 [74] 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  $StaticOffRoad(x)$ ,  $DynamicOffroad(x)$ ,  $FollowingVehicle(x)$ ,  $IrrelevantVehicle(x)$  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 concrete the high-level property in Eq 1 into the following format to describe the property in lane following scenario:

$$\begin{aligned} StaticCons(x) &:= (x.type == Static) \rightarrow (StaticOffRoad(x)) \\ VehicleCons(x) &:= (x.type == Vehicle) \rightarrow \\ &\quad (FollowVehicle(x) \vee IrrelevantVehicle(x)) \\ PedestrianCons(x) &:= (x.type == Pedestrian) \rightarrow \\ &\quad (DynamicOffroad(x) \vee StaticOffRoad(x)) \\ &\wedge_{x \in O} ((StaticCons(x)) \wedge (VehicleCons(x))) \\ &\wedge (PedestrianCons(x)) \rightarrow \neg Stop \end{aligned} \quad (2)$$

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 concreting 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 Appendix B. 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

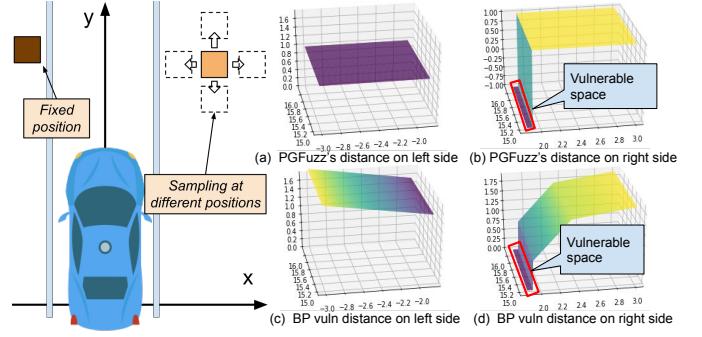


Figure 6: Illustration of the benefit of our BP vulnerability distance design compared to other fitness function choices.

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 [92], 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 [92]. 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 Appendix C.

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 [92], 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, 79, 80] 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 [93, 94], 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 [32], while Autoware is adopted by the USDOT in their AD vehicle fleet [33].

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 [95]. We did not evaluate on 6.0, the latest Apollo version, as it only

made minor changes to 5.0 in BP [95]. 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, 96], the latest version with an implementation of the open planner [97]. 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, an production-grade AD simulator [98], 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 [99–101]. LGSVL itself is also designed for performance and safety testing of production AD systems [98].

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. More details are in Table V in Appendix. 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 [102]); 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 Table VI in Appendix. 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.

Table II: Discovered BP DoS vulnerabilities from Apollo (3.0, 5.0) and Autoware. PI identifiers refer to PIs in Table IV in Appendix. All the vulnerabilities discovered in Apollo 5.0 are confirmed to also exist in the latest version Apollo 6.0.

Vuln #	Driving Scenario	Software	Violated PI #	Attack-influenced Planning Behavior	Triggering Objects
V1	Lane following	Apollo 3.0/5.0	PI1 (PI-C1)	Permanent stop	Static obstacle
V2	Lane changing	Apollo 3.0/5.0	PI3 (PI-C2, 3)	Fail to change lane and never reach destination	Vehicle
V3	Lane borrow	Apollo 3.0/5.0	PI4 (SP-PI-C1, 2)	Fail to borrow the lane and permanent stop	Vehicle or static obstacles in front of blocking vehicle
V4	Lane borrow	Apollo 3.0/5.0	PI4 (PI-C1, 4, 5)	Fail to borrow the lane and permanent stop	Off-road static obstacle
V5	Intersection w/ traffic signal	Apollo 3.0/5.0	PI6 (PI-C4)	Fail to pass intersection and permanent stop	Static pedestrian
V6	Intersection w/ traffic signal	Apollo 3.0/5.0	PI6 (PI-C5)	Emergency stop; possible to cause permanent stop and thus fail to pass intersection	Pedestrian who is leaving the intersection
V7	Intersection w/ stop sign	Apollo 3.0/5.0	PI5 (PI-C1)	Fail to pass intersection and permanent stop	Static bicycle off the road
V8	Lane following	Autoware	PI1 (PI-C1)	Permanent stop	Static obstacle off the lane
V9	Lane following	Autoware	PI1 (PI-C3)	Emergency stop	Moving vehicle off the lane

**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 is *unique* if its code-level decision logic (branches) that causes such vulnerability (e.g., those identified in §III-A) is different from the others, which is similar to unique crashes in traditional fuzzers [103–106]). 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. All experiments are run on AMD EPYC 7551 CPUs.

### 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; some 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 Appendix D.

**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, or 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. In this scenario, 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-lane or multiple-lane road). As discussed in §III-A, the root cause of such vulnerabilities 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 narrow urban road (e.g., 2.7m in width [69]) 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 belongs 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 place 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 borrowing behaviors. As the 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 and V3 are caused by overly-conservative design when checking if the AD vehicle’s future driving path overlaps with other vehicles during the lane changing or borrowing. Although PlanFuzz mainly aims to find BP DoS vulnerabilities introduced by overly-conservative planning decisions, V4 is in fact due to an implementation bug in the BP when it checks whether the perception range is blocked by any obstacles before performing lane borrowing. More details of the vulnerabilities can be found in Appendix D-C.

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 causes significant delays for the AD vehicle to reach its destination if the attacker keeps performing such an 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,  $\text{PlanFuzz}^{-\text{guide}}$ , 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,  $\text{PlanFuzz}^{-\text{PI}}$ , 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  $\text{PB-M}$ ). Protobuf-mutator is a readily-available fuzzer designed specifically for the data structure of protobufs [107], 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 [108]. For each fuzzer setup, we measure the discovered unique vulnerabilities (defined in §V-A), and their average Time-to-Exposure ( $\mu\text{TTE}$ ), 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 [92, 108, 109]. Specifically, we use the *Vargha-Delaney statistic*  $\hat{A}_{12}$  to measure the effect size, and use *Mann-Whitney U* [109] to measure the statistical significance of the  $\mu\text{TTE}$  performance drop, which is the recommended measure for assessing randomized algorithms like fuzzers [92, 109].

**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  $57\times$  less efficient/effective. This concretely shows that our two main problem-specific fuzzer designs, BP vulnerability distance (§IV-E) and PI-aware physical-object generation (§IV-D), are *necessary* for effectively discovering BP DoS vulnerabilities (RQ3).

For  $\text{PlanFuzz}^{-\text{guide}}$  and  $\text{PlanFuzz}^{-\text{PI}}$ , the setups that each still retains one of these two problem-specific designs, the same set of unique vulnerabilities (as the 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  $\text{PlanFuzz}^{-\text{guide}}$ , and 8/9 for  $\text{PlanFuzz}^{-\text{PI}}$ ), the  $\mu\text{TTE}$  performance drops are *statistically significant* (bold  $\hat{A}_{12}$  values in Table III). From the  $\mu\text{TTE}$  values,  $\text{PlanFuzz}^{-\text{guide}}$  and  $\text{PlanFuzz}^{-\text{PI}}$  are on average  $>4.5\times$  and  $>3.5\times$  slower respectively; for complicated cases such as V1, such degradation can be even over  $7.7\times$  for  $\text{PlanFuzz}^{-\text{guide}}$ . Among the 9 vulnerabilities, V4 is the only one that does not show significant  $\mu\text{TTE}$  differences for both  $\text{PlanFuzz}^{-\text{guide}}$  and  $\text{PlanFuzz}^{-\text{PI}}$ ; this is because it is relatively easier to trigger by nature due to a likely range-checking bug (Appendix §D-D), which can be seen by its much lower  $\mu\text{TTE}$  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 as shown in Table III.

Table III: Results of baseline comparison. PlanFuzz<sup>-guide</sup>: PlanFuzz without BP vulnerability distance as guidance. PlanFuzz<sup>-PI</sup>: PlanFuzz without PI-aware physical-object generation. PB-M: Protobuf-mutator (a directly-compatible traditional fuzzer). Each setup is run 10 times, 24h each time.  $\mu\text{TTE}$  is average Time-To-Exposure. NF: Vulnerability not found in 24h. **Bold**  $\hat{A}_{12}$  values denote statistically significant  $\mu\text{TTE}$  performance drops.

Seed	Uniq. vuln	PlanFuzz $\mu\text{TTE}$	PlanFuzz <sup>-guide</sup> $\mu\text{TTE}$	$\hat{A}_{12}$	PlanFuzz <sup>-PI</sup> $\mu\text{TTE}$	$\hat{A}_{12}$	PB-M $\mu\text{TTE}$
1	V1	165s	1278s (7.74 $\times$ )	<b>0.97</b>	276s (1.67 $\times$ )	<b>0.88</b>	NF
2	V2	19s	21s (1.11 $\times$ )	0.55	117s (6.15 $\times$ )	<b>0.98</b>	NF
3	V3	16s	34s (2.12 $\times$ )	<b>0.88</b>	53s (3.31 $\times$ )	<b>0.88</b>	NF
	V4	1s	1s (1.00 $\times$ )	0.57	2s (2.00 $\times$ )	0.58	NF
4	V5	47s	92s (1.95 $\times$ )	<b>0.89</b>	167s (3.55 $\times$ )	<b>0.98</b>	NF
	V6	35s	78s (2.22 $\times$ )	<b>0.83</b>	148s (4.22 $\times$ )	<b>0.93</b>	NF
5	V7	45s	119s (2.64 $\times$ )	<b>0.97</b>	208s (4.62 $\times$ )	<b>0.96</b>	NF
6	V8	53s	193s (3.64 $\times$ )	<b>0.96</b>	327s (6.16 $\times$ )	<b>0.90</b>	NF
7	V9	37s	57s (1.54 $\times$ )	<b>0.93</b>	188s (5.08 $\times$ )	<b>0.96</b>	NF
	Average	46s	208s (4.52 $\times$ )	<b>0.83</b>	165s (3.58 $\times$ )	<b>0.89</b>	NF

## 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 [98], and simulate with the complete AD stack (Apollo or Autoware) in the loop more than 10 times, which also include other AD system components such as localization, perception, and control. We create attack demos for all the case studies listed in this section. Demo videos are available at our project website <https://sites.google.com/view/cav-sec/planfuzz> [34].

### 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` [110] (pseudo code can be found in Fig. 26 in the Appendix). 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 [69]), 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 [69], 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  $\sim$ 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  $\sim$ 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 [34] 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 [34]. 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 [111] equipped with Velodyne VLP-32C LiDAR [112] and NovAtel Positioning Kits [113]. 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 [114, 115] to get the object detection results and launch the op\_planner [97] 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 come to the conclusion that the Autoware’s lane following vulnerability can be reconstructed in real world. Fig. 7 shows our experiment setup and trace visualization 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. As shown in Fig. 24 [116], 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 [69]), 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 (illustrated in Fig. 25). Here, 2 parked bicycles are enough to bypass all timeout mechanisms in Apollo BP (detailed in Appendix D-G). 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` [117]. 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  $\leq 3.55\text{m}$  width (applicable to almost all common local and highway lanes (up to 3.6m wide) [69]), 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 ( $>70\text{cm}$ )) towards the direction she wants to prevent the victim from changing lane to.

**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

From the vulnerability discussions in §V-B and Appendix D), 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).

**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 [118] (which is one reason why highway lanes are wider than local ones [119]). However, how to design such dynamic adjustment is also 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. Besides improving the decision parameter/logic, another promising direction for future work is to detect such attacks that exploit conservative designs, e.g., detecting the placement patterns of road-side static obstacles in V1.

### B. Limitations and Future Work

**Tool effectiveness.** PlanFuzz's results do not have false positives since all the discovered vulnerabilities have been

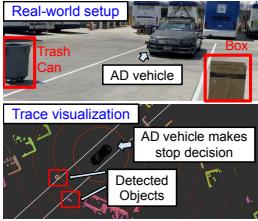


Figure 7: Real-world experiment setup (top) and sensor trace visualization (bottom).

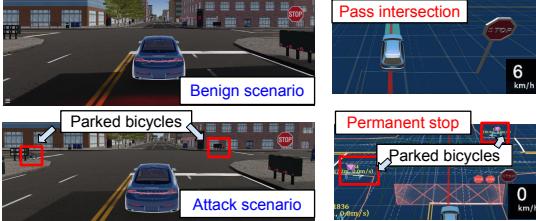


Figure 8: Benign (top): AD vehicle passes the intersection. Attack (bottom): AD vehicle permanently stops because of the parked bicycles placed by the attacker.

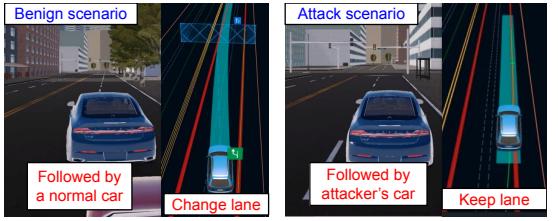


Figure 9: Benign (left): Successfully changes lanes even if it is followed by another vehicle. Attack (right): Fails to change lanes due to the attacker's vehicle.

```

1 # Iterate over the obstacle list
2 for (each obs : obstacle_list):
3     # Judge based on the lateral position of a veh.
4     # start_l, end_l are veh's left/right boundaries
5     if (obs.end_l < -2.5 or obs.start_l > 2.5):
6         continue
7     # The backward safe buffer
8     BackwardSafeBuffer ← 4.0f
9     # Check whether the veh is close
10    if (ego_start_s-obs.end_s<BackwardSafeBuffer):
11        IsClearChangeLane ← false

```

Figure 10: Simplified vulnerable pseudo code of V2 (lane changing).

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, FN 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.

In addition, similar to other dynamic testing approaches, the output of our current design is a set of concrete test inputs that can trigger a discovered vulnerability. However, this can only provide several example triggering placements of the physical objects, not the complete *triggering ranges* of such placements, which can be more informative. To achieve the latter, more sophisticated formal methods such as symbolic execution [120] can be leveraged, but such methods are known to generally suffer from scalability limitations when applied to real-world code bases [121], not to mention that in our problem settings one needs to further overcome the semantic gaps by systematically locating and extracting the physical object-related code-level constraints. We thus leave a systematic exploration of this direction to future work.

**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 [122], 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 in this paper, adding road objects (e.g., carbon boxes, bikes, vehicles) is not the only possible threat model. For example, attackers may also attack the perceptual sensors, such as by sensor spoofing [7] or compromising internal AD system components [123], to introduce malicious inputs to BP. Since these attack vectors may have stronger BP input perturbation capabilities/flexibility, more BP DoS vulnerabilities may be discovered. 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 [123], 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, 124–127]. 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, 128–138]. 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 context [139, 140]. Some developed methods to discover semantic vulnerabilities in the DNN models in AD [7, 9, 23, 24, 128–131, 134, 135, 141, 142]. 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, 76–80, 143]. They can not 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 systems, RVs typically follow the control commands sent by a base station without the need to make planning decisions by itself based on the surrounding environment. Thus, existing works concentrate on designs to find control-specific vulnerabilities [25, 26, 28] or highly rely on control-specific knowledge [27, 138], which are thus orthogonal to the design challenges we need to address for discovering semantic vulnerabilities for BP in the AD context.

**Traditional fuzzers.** There are various traditional fuzzers [103, 144] that can automatically discover generic software vulnerabilities such as software crash and memory corruptions. However, they cannot readily be used for our problem setting since they need to at least change the testing oracle from generic software symptoms (e.g., crashes) to the semantic ones (overly-conservative planning decisions). For the latter, a valid and systematic definition is the very first design challenge that needs to be overcome (§III-B). However, that is still not enough since to achieve effective discovery, they further need to add new problem-specific input generation and guidance designs such as those we designed (§IV-E, §IV-D); otherwise, they still cannot discover any such vulnerabilities even if given 24h as shown in §V-C.

## 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 BP implementations from practical AD systems, and find that it can effectively discover 9 previously-unknown semantic DoS vulnerabilities without false positives. We further perform exploitation case studies using simulation and real-vehicle traces to validate their realism and impacts. We also 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

- [1] S. O.-R. A. V. S. Committee *et al.*, “Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles,” SAE International: Warrendale, PA, USA, 2018.
- [2] “Waymo is Opening its Fully Driverless Service to the General Public in Phoenix.” <https://blog.waymo.com/2020/10/waymo-is-opening-its-finally-driverless.html>.
- [3] “UPS Joins Race for Future of Delivery Services by Investing in Self-driving Trucks.” <https://abcnews.go.com/Business/ups-joins-race-future-delivery-services-investing-driving/story?id=65014414>.
- [4] “PonyAI Robotaxi Service.” <https://www.pony.ai/en/index.html>.
- [5] J. Shen, J. Y. Won, and Q. A. Chen, “Drift with Devil: Security of Multi-Sensor Fusion based Localization in High-Level Autonomous Driving under GPS Spoofing,” in *Usenix Security*, 2020.
- [6] C. Yulong, W. Ningfei, X. Chaowei, Y. Dawei, F. Jin, Y. Ruigang, C. Qi Alfred, L. Mingyan, and L. Bo, “Invisible for both Camera and LiDAR: Security of Multi-Sensor Fusion based Perception in Autonomous Driving Under Physical-World Attacks,” in *42nd IEEE Symposium on Security and Privacy (IEEE S&P)*, 2021.
- [7] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, “Adversarial Sensor Attack on LiDAR-based Perception in Autonomous Driving,” in *ACM CCS*, 2019.
- [8] J. Sun, Y. Cao, Q. A. Chen, and Z. M. Mao, “Towards robust lidar-based perception in autonomous driving: General black-box adversarial sensor attack and countermeasures,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 877–894, 2020.
- [9] Y. Zhao, H. Zhu, R. Liang, Q. Shen, S. Zhang, and K. Chen, “Seeing isn’t Believing: Practical Adversarial Attack Against Object Detectors,” in *ACM CCS*, 2019.
- [10] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta, “Formal Scenario-based Testing of Autonomous Vehicles: From Simulation to the Real World,” in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1–8, IEEE, 2020.
- [11] “Death of Elaine Herzberg.” [https://en.wikipedia.org/wiki/Death\\_of\\_Elaine\\_Herzberg](https://en.wikipedia.org/wiki/Death_of_Elaine_Herzberg).
- [12] “Tesla In Taiwan Crashes Directly Into Overturned Truck, Ignores Pedestrian, With Autopilot On.” <https://www.forbes.com/sites/bradtempleton/2020/06/02/tesla-in-taiwan-crashes-directly-into-overturned-truck-ignores-pedestrian-with-autopilot-on/?sh=76e79dd258e5>.
- [13] “Two People Killed in Fiery Tesla Crash with No One Driving.” <https://www.theverge.com/2021/4/18/22390612/two-people-killed-fiery-tesla-crash-no-driver>.
- [14] “Tesla on Autopilot Crashes into Michigan Cop’s Patrol Car.” <https://nypost.com/2021/03/18/tesla-on-autopilot-crashes-into-michigan-troopers-patrol-car>.
- [15] “Waymo Vehicle Gets Stuck, Then Abandons Me (Also I found a UI bug!) | JJRicks Rides with Waymo 27.” <https://youtu.be/SH7cQoJIT8>.
- [16] “Waymo Self Driving: TWO Roadside Assistance Takeovers and MUCH More! | JJRicks Rides With Waymo 21.” <https://youtu.be/Uy3DuUOLhUE>.
- [17] “Waymo Self Driving Taxi Fumbles In Construction Zone, Blocks Traffic | JJRicks Rides with Waymo 54.” <https://youtu.be/zdKCQKBvH-A>.
- [18] “Humans Will Bully Mild-Mannered Autonomous Cars.” <https://www.technologyreview.com/2016/11/03/156270/humans-will-bully-mild-mannered-autonomous-cars>.
- [19] “Uber Says People are Bullying its Self-driving Cars with Rude Gestures and Road Rage.” <https://www.businessinsider.com/uber-people-bullying-self-driving-cars-2019-6>.
- [20] “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>.
- [21] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, “Nezha: Efficient Domain-independent Differential Testing,” in *2017 IEEE Symposium on security and privacy (SP)*, pp. 615–632, IEEE, 2017.
- [22] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, “Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations,” in *2014 IEEE Symposium on Security and Privacy*, pp. 114–129, IEEE, 2014.
- [23] K. Pei, Y. Cao, J. Yang, and S. Jana, “Deepxplore: Automated White-box Testing of Deep Learning Systems,” in *SOSP*, 2017.
- [24] Y. Tian, K. Pei, S. Jana, and B. Ray, “Deeptest: Automated Testing of Deep-neural-network-driven Autonomous Cars,” in *ICSE*, 2018.
- [25] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu, “RVFUZZER: Finding Input Validation Bugs in Robotic Vehicles through Control-guided Testing,” in *USENIX Security*, 2019.
- [26] H. Choi, S. Kate, Y. Aafer, X. Zhang, and D. Xu, “Cyber-Physical Inconsistency Vulnerability Identification for Safety Checks in Robotic Vehicles,” in *ACM CCS*, 2020.
- [27] T. Kim, C. H. Kim, A. Ozen, F. Fei, Z. Tu, X. Zhang, X. Deng, D. J. Tian, and D. Xu, “From Control Model to Program: Investigating

- gating Robotic Aerial Vehicle Accidents with MAYDAY,” in *USENIX Security*, 2020.
- [28] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, “PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles,” in *NDSS*, 2021.
- [29] “Apollo: An Open Autonomous Driving Platform.” <https://github.com/ApolloAuto/apollo>.
- [30] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, “Autoware On Board: Enabling Autonomous Vehicles with Embedded Systems,” in *ICCPs’18*, pp. 287–296, IEEE Press, 2018.
- [31] “Navigant Research Names Waymo, Ford Autonomous Vehicles, Cruise, and Baidu the Leading Developers of Automated Driving Systems.” <https://www.businesswire.com/news/home/20200407005119/en/Navigant-Research-Names-Waymo-Ford-Autonomous-Vehicles>.
- [32] “Baidu launches Apollo Go Robotaxi service in Beijing.” <https://www.prnewswire.com/news-releases/baidu-launches-apollo-go-robotaxi-service-in-beijing-advancing-its-leading-position-in-autonomous-driving-301127496.html>.
- [33] “Carma Platform.” <https://github.com/usdot-fhwa-stol/carma-platform>.
- [34] “PlanFuzz Website.” <https://sites.google.com/view/cav-sec/planfuzz>.
- [35] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, “A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles,” *IEEE Transactions on intelligent vehicles*, vol. 1, no. 1, pp. 33–55, 2016.
- [36] M. Ilievski, S. Sedwards, A. Gaurav, A. Balakrishnan, A. Sarkar, J. Lee, F. Bouchard, R. De Iaco, and K. Czarnecki, “Design Space of Behaviour Planning for Autonomous Driving,” *arXiv preprint arXiv:1908.07931*, 2019.
- [37] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al., “Autonomous Driving in Urban Environments: Boss and the Urban Challenge,” *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.
- [38] M. Buehler, K. Iagnemma, and S. Singh, *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*, vol. 56, Springer, 2009.
- [39] D. González, J. Pérez, V. Milanés, and F. Nashashibi, “A Review of Motion Planning Techniques for Automated Vehicles,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 4, pp. 1135–1145, 2015.
- [40] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, and M. H. Ang, “Perception, Planning, Control, and Coordination for Autonomous Vehicles,” *Machines*, vol. 5, no. 1, p. 6, 2017.
- [41] W. Schwarting, J. Alonso-Mora, and D. Rus, “Planning and Decision-making for Autonomous Vehicles,” *Annual Review of Control, Robotics, and Autonomous Systems*, 2018.
- [42] L. Sun, C. Peng, W. Zhan, and M. Tomizuka, “A Fast Integrated Planning and Control Framework for Autonomous Driving via Imitation Learning,” in *Dynamic Systems and Control Conference*, vol. 51913, p. V003T37A012, American Society of Mechanical Engineers, 2018.
- [43] Y. Chen, P. Praveen, M. Priyantha, K. Muelling, and J. Dolan, “Learning on-road Visual Control for Self-driving Vehicles with Auxiliary Tasks,” in *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 331–338, IEEE, 2019.
- [44] P. Abbeel, D. Dolgov, A. Y. Ng, and S. Thrun, “Apprenticeship Learning for Motion Planning with Application to Parking Lot Navigation,” in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1083–1090, IEEE, 2008.
- [45] J. Koutník, G. Cuccu, J. Schmidhuber, and F. Gomez, “Evolving Large-Scale Neural Networks for Vision-Based Reinforcement Learning,” in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO ’13, (New York, NY, USA), p. 1061–1068, Association for Computing Machinery, 2013.
- [46] S. Wang, D. Jia, and X. Weng, “Deep Reinforcement Learning for Autonomous Driving,” *arXiv preprint arXiv:1811.11329*, 2018.
- [47] D. Loiacono, A. Prete, P. L. Lanzi, and L. Cardamone, “Learning to Overtake in TORCS using Simple Reinforcement Learning,” in *IEEE Congress on Evolutionary Computation*, pp. 1–8, IEEE, 2010.
- [48] E. Capo and D. Loiacono, “Short-Term Trajectory Planning in TORCS using Deep Reinforcement Learning,” in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 2327–2334, IEEE, 2020.
- [49] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, “Deep Reinforcement Learning Framework for Autonomous Driving,” *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.
- [50] B. Osifński, A. Jakubowski, P. Zięcina, P. Miłoś, C. Galias, S. Homoceanu, and H. Michalewski, “Simulation-based reinforcement learning for real-world autonomous driving,” in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6411–6418, 2020.
- [51] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah, “Learning to Drive in a Day,” in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 8248–8254, IEEE, 2019.
- [52] A. Kuefler, J. Morton, T. Wheeler, and M. Kochenderfer, “Imitating Driver Behavior with Generative Adversarial Networks,” in *2017 IEEE Intelligent Vehicles Symposium (IV)*, pp. 204–211, IEEE, 2017.
- [53] J. S. H. S. T. Han and B. Zhou, “Neuro-Symbolic Program Search for Autonomous Driving Decision Module Design,” in *Proceedings of the 2020 Conference on Robot Learning*, 2020.
- [54] H. Bai, D. Hsu, and W. S. Lee, “Integrated Perception and Planning in the Continuous Space: A POMDP Approach,” *The International Journal of Robotics Research*, vol. 33, no. 9, pp. 1288–1302, 2014.
- [55] S. Brechtel, T. Gindelé, and R. Dillmann, “Solving Continuous POMDPs: Value Iteration with Incremental Learning of an Efficient Space Representation,” in *International Conference on Machine Learning*, pp. 370–378, PMLR, 2013.
- [56] K. H. Wray, S. J. Witwicki, and S. Zilberstein, “Online Decision-making for Scalable Autonomous Systems,” in *International Joint Conference on Artificial Intelligence*, 2017.
- [57] F. Behbahani, K. Shiarlis, X. Chen, V. Kurin, S. Kasewa, C. Stirbu, J. Gomes, S. Paul, F. A. Oliehoek, J. Messias, et al., “Learning from Demonstration in the Wild,” in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 775–781, IEEE, 2019.
- [58] L. Chi and Y. Mu, “Deep Steering: Learning end-to-end Driving Model from Spatial and Temporal Visual Cues,” *arXiv preprint arXiv:1708.03798*, 2017.
- [59] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, “A Survey of Autonomous Driving: Common Practices and Emerging Technologies,” *IEEE Access*, vol. 8, pp. 58443–58469, 2020.
- [60] “Path Planning Project Udacity Self-driving Car Nanodegree.” <https://medium.com/intro-to-artificial-intelligence/path-planning-project-udacitys-self-driving-car-nanodegree-be1f531cc4f7>.
- [61] “Argo Developing a Self-Driving System You Can Trust.” <https://www.argo.ai/wp-content/uploads/2021/04/ArgoSafetyReport.pdf>.
- [62] “Tesla Under Investigation by SEC after Fatal Crash Involving Autopilot Report.” <https://www.theguardian.com/technology/2016/jul/11/sec-tesla-self-driving-car-crash-death-investigation>.
- [63] “Aurora Acquires Uber’s Automated Driving Unit—And Uber’s Cash.” <https://www.forbes.com/sites/samabuelsamid/2020/12/07/aurora-acquires-ubers-automated-driving-unitand-ubers-cash/?sh=3ba6dbfa63eb>.
- [64] “Baidu Will Launch Driverless Robotaxis In Beijing This Week.” <https://tinyurl.com/twd5zbn6>.
- [65] “Waymo is opening its fully driverless service to the general public in Phoenix.” <https://blog.waymo.com/2020/10/waymo-is-opening-its-fully-driverless.html>.
- [66] “Nuro can now operate and charge for autonomous delivery services in California.” <https://tinyurl.com/2trxzjre>.
- [67] “Traffic Jams Cost U.S. Drivers \$ 1,200 a Year: Study.” <https://www.reuters.com/article/us-autos-congestion/traffic-jams-cost-u-s-drivers-1200-a-year-study-idUSKBN15Z0DL>.
- [68] Q. A. Chen, Y. Yin, Y. Feng, Z. M. Mao, and H. X. Liu, “Exposing congestion attack on emerging connected vehicle based traffic signal control,” in *NDSS*, 2018.
- [69] “USDOT Lane Width.” [https://safety.fhwa.dot.gov/geometric/pubs/mitigationstrategies/chapter3/3\\_lanewidth.cfm](https://safety.fhwa.dot.gov/geometric/pubs/mitigationstrategies/chapter3/3_lanewidth.cfm).
- [70] “Automotive Platforms.” <https://autonomoustuff.com/products/astuff-automotive>.
- [71] “Ford Fusion / Fusion Hybrid Features And Specs.” <https://www.caranddriver.com/ford/fusion/specs>.
- [72] “Lincoln MKZ Features And Specs.” <https://www.caranddriver.com/lincoln/mkz/specs>.
- [73] “Jaguar I-Pace Wiki.” [https://en.wikipedia.org/wiki/Jaguar\\_I-Pace](https://en.wikipedia.org/wiki/Jaguar_I-Pace).
- [74] “California Driver Handbook.” <https://www.dmv.ca.gov/portal/handbook/california-driver-handbook/laws-and-rules-of-the-road>.
- [75] “Vehicle Code 22500 CVC – Improper Parking, Stopping or Standing.” <https://www.shouselaw.com/ca/defense/vehicle-code/22500>.

- [76] D. J. Fremont, J. Chiu, D. D. Margineantu, D. Osipychev, and S. A. Seshia, "Formal Analysis and Redesign of a Neural Network-based Aircraft Taxiing System with VerifAI," in *International Conference on Computer Aided Verification*, pp. 122–134, Springer, 2020.
- [77] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, "Verifai: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-based Systems," in *CAV*, 2019.
- [78] T. Dreossi, A. Donzé, and S. A. Seshia, "Compositional Falsification of Cyber-physical Systems with Machine Learning Components," *Journal of Automated Reasoning*, vol. 63, no. 4, pp. 1031–1053, 2019.
- [79] M. Hekmatnejad, B. Hoxha, and G. Fainekos, "Search-based Test-case Generation by Monitoring Responsibility Safety Rules," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1–8, IEEE, 2020.
- [80] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, "Simulation-based adversarial test generation for autonomous vehicles with machine learning components," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1555–1562, IEEE, 2018.
- [81] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: A Language for Scenario Specification and Scene Generation," in *PLDI*, 2019.
- [82] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *ACM CCS, CCS '18*, 2018.
- [83] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," in *ACM CCS*, 2018.
- [84] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart Greybox Fuzzing," *IEEE Transactions on Software Engineering*, 2019.
- [85] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," *IEEE Transactions on Software Engineering*, 2017.
- [86] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic Greybox Fuzzing for Structured Binary Formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1–13, 2020.
- [87] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *NDSS*, vol. 17, pp. 1–14, 2017.
- [88] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 477–486, IEEE, 2007.
- [89] S. Rawat and L. Mounier, "An Evolutionary Computing Approach for Hunting Buffer Overflow Vulnerabilities: A Case of Aiming in Dim Light," in *2010 European Conference on Computer Network Defense*, IEEE, 2010.
- [90] D. Whitley, "A Genetic Algorithm Tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.
- [91] "General Motors 2018 Self-Driving Safety Report," <https://www.gm.com/content/dam/company/docs/us/en/gmcom/gmsafetyreport.pdf>.
- [92] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, 2017.
- [93] "Autoware Self-driving Vehicle on a Highway." [https://www.youtube.com/watch?v=npQMzH3j\\_d8](https://www.youtube.com/watch?v=npQMzH3j_d8).
- [94] "Baidu Launches Their Open Platform for Autonomous Cars and We Got to Test it." <https://technode.com/2017/07/05/baidu-apollo-1-0-autonomous-cars-we-test-it/>.
- [95] "Baidu Apollo." <https://github.com/ApolloAuto/apollo/releases>.
- [96] "Autoware.AI." <https://www.autoware.ai>.
- [97] H. Darweesh, E. Takeuchi, K. Takeda, Y. Ninomiya, A. Sujiwo, L. Y. Morales, N. Akai, T. Tomizawa, and S. Kato, "Open-source Integrated Planner for Autonomous Navigation in Highly Dynamic Environments," *Journal of Robotics and Mechatronics*, vol. 29, no. 4, pp. 668–684, 2017.
- [98] LG, "LGSVL Simulator: An Autonomous Vehicle Simulator." <https://github.com/lgsvl/simulator>.
- [99] "Simulation Becomes Increasingly Important For Self-Driving Cars." <https://www.forbes.com/sites/davidsilver/2018/11/01/simulation-becomes-increasingly-important-for-self-driving-cars/>.
- [100] "Inside Waymo's Secret World for Training Self-Driving Cars." <https://www.theatlantic.com/technology/archive/2017/08/inside-waymos-secret-testing-and-simulation-facilities/537648/>.
- [101] "Simulation Engine from MIT Trains Self-Driving Cars Before They Hit Real Roads." <https://www.therobotreport.com/simulation-engine-mit-trains-self-driving-cars-before-hit-real-streets/>.
- [102] "Tesla's Sensor Coverage (Almost 306 Degree Coverage at 80m)." <https://www.tesla.com/autopilot>.
- [103] "AFL." <http://lcamtuf.coredump.cx/afl>.
- [104] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *USENIX Security*, 2018.
- [105] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711–725, IEEE, 2018.
- [106] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshtaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *NDSS*, 2016.
- [107] "Protobuf-Mutator." <https://github.com/google/libprotobuf-mutator>.
- [108] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, 2018.
- [109] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1–10, IEEE, 2011.
- [110] "Autoware Trajectory Evaluator." [https://github.com/Autoware-AI/core\\_planning/blob/master/op\\_local\\_planner/nodes/op\\_trajectory\\_evaluator/op\\_trajectory\\_evaluator.cpp](https://github.com/Autoware-AI/core_planning/blob/master/op_local_planner/nodes/op_trajectory_evaluator/op_trajectory_evaluator.cpp).
- [111] "Lincoln MKZ Datasheet." <https://autonomoustuff.com/-/media/Images/Hexagon/Hexagon%20Core/autonomoustuff/pdf/as-lincoln-mkz-datasheet.ashx>.
- [112] "Velodyne LiDAR's ULTRA Puck VLP-32C." [https://hexagondownloads.blob.core.windows.net/public/AutonomouStuff/wp-content/uploads/2019/05/VLP-32C\\_Ultra\\_Puck-whitelabel.pdf](https://hexagondownloads.blob.core.windows.net/public/AutonomouStuff/wp-content/uploads/2019/05/VLP-32C_Ultra_Puck-whitelabel.pdf).
- [113] "NovAtel Positioning Kits." <https://hexagondownloads.blob.core.windows.net/public/AutonomouStuff/wp-content/uploads/2019/11/pwrrpak7d-e2-product-sheet.pdf>.
- [114] "Autoware.AI LiDAR Euclidean Cluster Detect." [https://github.com/Autoware-AI/core\\_perception/tree/master/lidar\\_euclidean\\_cluster\\_detect](https://github.com/Autoware-AI/core_perception/tree/master/lidar_euclidean_cluster_detect).
- [115] "Autoware LiDAR Kf Contour Track." [https://github.com/Autoware-AI/core\\_perception/tree/master/lidar\\_kf\\_contour\\_track](https://github.com/Autoware-AI/core_perception/tree/master/lidar_kf_contour_track).
- [116] "Apollo Stop Sign Scenario." [https://github.com/ApolloAuto/apollo/blob/master/modules/planning/scenarios/stop\\_sign/unprotected\\_stage\\_pre\\_stop.cc](https://github.com/ApolloAuto/apollo/blob/master/modules/planning/scenarios/stop_sign/unprotected_stage_pre_stop.cc).
- [117] "Apollo Lane Change Decider." [https://github.com/ApolloAuto/apollo/blob/master/modules/planning/tasks/deciders/lane\\_change\\_decider/lane\\_change\\_decider.cc](https://github.com/ApolloAuto/apollo/blob/master/modules/planning/tasks/deciders/lane_change_decider/lane_change_decider.cc).
- [118] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "On a Formal Model of Safe and Scalable Self-Driving Cars," *arXiv preprint arXiv:1708.06374*, 2017.
- [119] "Urban Street Design Guide." <https://nacto.org/publication/urban-street-design-guide/street-design-elements/lane-width/>.
- [120] C. Cedar, D. Dunbar, D. R. Engler, et al., "Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *OSDI*, vol. 8, pp. 209–224, 2008.
- [121] D. A. Ramos and D. Engler, "Under-Constrained Symbolic Execution: Correctness Checking for Real Code," in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 49–64, 2015.
- [122] "LLVM Project." <https://llvm.org/>.
- [123] S. Jha, S. Cui, S. Banerjee, J. Cyriac, T. Tsai, Z. Kalbarczyk, and R. K. Iyer, "MI-Driven Malware that Targets AV Safety," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 113–124, IEEE, 2020.
- [124] C. Yan, W. Xu, and J. Liu, "Can You Trust Autonomous Vehicles: Contactless Attacks against Sensors of Self-driving Vehicle," *DEF CON*, vol. 24, no. 8, p. 109, 2016.
- [125] B. Nassi, D. Nassi, R. Ben-Netanel, Y. Mirsky, O. Drokin, and Y. Elovici, "Phantom of the ADAS: Phantom Attacks on Driver-Assistance Systems," in *IACR*, 2020.
- [126] H. Shin, D. Kim, Y. Kwon, and Y. Kim, "Illusion and Dazzle: Adversarial Optical Channel Exploits Against LiDARs for Automotive

- Applications,” in *International Conference on Cryptographic Hardware and Embedded Systems*, pp. 445–467, Springer, 2017.
- [127] Y. Tu, Z. Lin, I. Lee, and X. Hei, “Injected and Delivered: Fabricating Implicit Control over Actuation Systems by Spoofing Inertial Sensors,” in *USENIX Security*, pp. 1545–1562, 2018.
- [128] T. Sato, J. Shen, N. Wang, Y. Jia, X. Lin, and Q. A. Chen, “Dirty Road Can Attack: Security of Deep Learning based Automated Lane Centering under Physical-World Attack,” in *USENIX Security*, 2021.
- [129] T. Sato, J. Shen, N. Wang, Y. J. Jia, X. Lin, and Q. A. Chen, “Deployability Improvement, Stealthiness User Study, and Safety Impact Assessment on Real Vehicle for Dirty Road Patch Attack,” in *AutoSec Workshop at NDSS*, 2021.
- [130] Y. Cao, N. Wang, C. Xiao, D. Yang, J. Fang, R. Yang, Q. A. Chen, M. Liu, and B. Li, “Invisible for both Camera and LiDAR: Security of Multi-Sensor Fusion based Perception in Autonomous Driving Under Physical World Attacks,” in *IEEE S&P*, 2021.
- [131] H. Liang, R. Jiao, T. Sato, J. Shen, Q. A. Chen, and Q. Zhu, “End-to-End Analysis of Adversarial Attacks to Automated Lane Centering Systems,” in *AutoSec Workshop at NDSS*, 2021.
- [132] K. Tang, J. Shen, and Q. A. Chen, “Fooling Perception via Location: A Case of Region-of-Interest Attacks on Traffic Light Detection in Autonomous Driving,” in *AutoSec Workshop at NDSS*, 2021.
- [133] Y. Jia, Y. Lu, J. Shen, Q. A. Chen, H. Chen, Z. Zhong, and T. Wei, “Fooling Detection Alone is Not Enough: Adversarial Attack Against Multiple Object Tracking,” in *ICLR*, 2019.
- [134] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, F. Tramer, A. Prakash, T. Kohno, and D. Song, “Physical Adversarial Examples for Object Detectors,” in *WOOT*, 2018.
- [135] S.-T. Chen, C. Cornelius, J. Martin, and D. H. P. Chau, “Shapeshifter: Robust Physical Adversarial Attack on Faster R-CNN Object Detector,” in *ECML PKDD*, 2018.
- [136] “Model Hacking ADAS to Pave Safer Roads for Autonomous Vehicles.” <https://www.mcafee.com/blogs/other-blogs/mcafee-labs/model-hacking-adas-to-pave-safer-roads-for-autonomous-vehicles/>, 2020.
- [137] R. Quinonez, J. Giraldo, L. Salazar, E. Bauman, A. Cardenas, and Z. Lin, “Savior: Securing autonomous vehicles with robust physical invariants,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 895–912, 2020.
- [138] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Deng, “Detecting Attacks against Robotic Vehicles: A control Invariant Approach,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 801–816, 2018.
- [139] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, “A Comprehensive Study of Autonomous Vehicle Bugs,” in *ICSE*, 2020.
- [140] D. K. Hong, J. Kloosterman, Y. Jin, Y. Cao, Q. A. Chen, S. Mahlke, and Z. M. Mao, “AVGuardian: Detecting and Mitigating Publish-Subscribe Overprivilege for Autonomous Vehicle Systems,” in *EuroS&P*, 2020.
- [141] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, “Deepbillboard: Systematic Physical-world Testing of Autonomous Driving Systems,” in *ICSE*, IEEE, 2020.
- [142] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, “DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems,” in *ASE*, IEEE, 2018.
- [143] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, “AV-FUZZER: Finding Safety Violations in Autonomous Driving Systems,” in *ISSRE*, IEEE, 2020.
- [144] “libfuzzer.” <https://llvm.org/docs/LibFuzzer.html>.
- [145] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The Program Dependence Graph and its Use in Optimization,” *TOPLAS*, 1987.
- [146] S. Liu, G. Tan, and T. Jaeger, “PtrSplit: Supporting General Pointers in Automatic Program Partitioning,” in *ACM CCS*, 2017.
- [147] “Automated Vehicles for Safety.” <https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety>.
- [148] “Code Comment in Apollo IsPerceptionBlocked.” [https://github.com/ApolloAuto/apollo/blob/master/modules/planning/tasks/deciders/lane\\_change\\_decider/lane\\_change\\_decider.h#L48](https://github.com/ApolloAuto/apollo/blob/master/modules/planning/tasks/deciders/lane_change_decider/lane_change_decider.h#L48).

## APPENDIX A SUMMARY OF PLANNING INVARIANT

We introduce the formal definition of constraints on physical 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:

$$(s, l, laneID) = transform(pos, m) \quad (3)$$

We define the function with input position  $pos$  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 ( $laneID$ ). 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  $x.polygon$ :

$$\begin{aligned} StaticOffRoad(x) := \\ \wedge_{p \in x.polygon} |transform(p, m).l| > halfLaneWidth \end{aligned} \quad (4)$$

This means every polygon point of the object boundary is not within the range of any lane. The  $halfLaneWidth$  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,  $x.polygons(w)$  for the polygon points at waypoint  $w$ , the constraints for the pedestrian can be defined as:

$$\begin{aligned} DynamicOffRoad(x) := \\ (\wedge_{w \in x.W} (\wedge_{p \in x.polygons(w)} |transform(p, m).l| > halfLaneWidth)) \\ \wedge (\text{innerProduct}(x.heading, directionTowardsLane) < 0) \end{aligned} \quad (5)$$

The property *DynamicOffRoad* will check all the waypoints in the future moving trajectory of a physical object  $x$  and make sure that the 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 following the AD vehicle and it should not affect the planning behavior. For simplicity, we also use a property *driveInLane*( $x, laneID$ ) to describe that the vehicle  $x$  keeps driving within the lane denoted by the lane ID ( $laneID$ ). This can be modeled as:

$$\begin{aligned} FollowVehicle(x) := \\ (transform(x.pos, m).laneID == ego.laneID) \\ \wedge (transform(x.pos, m).s + safetyFollowingDistance < ego.s) \\ \wedge (x.velocity \leq ego.velocity) \\ \wedge \text{driveInLane}(x, transform(x.pos, m).laneID) \end{aligned} \quad (6)$$

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

$$\begin{aligned} \text{IrrelevantVehicle}(x) := \\ (\text{transform}(x.\text{pos}, m).\text{laneID} \neq \text{ego}.\text{laneID}) \\ \wedge \text{driveInLane}(x, \text{transform}(x.\text{pos}, m).\text{laneID}) \end{aligned} \quad (7)$$

## APPENDIX B

### DETAILS OF PI CONSTRAINT ENFORCEMENT IN §IV-D

As described in §IV-D, in the PI-constraints enforcement we need to enforce the randomly-generated static properties to conform to PI constraints. As it is used in the mutation step of a genetic algorithm, such enforcement step needs to have the following proprieties:

- **Diversity.** A good mutation function in genetic algorithm should introduce randomness and diversity to the next generation such that the genetic algorithm can explore different descent directions and avoid local minima.
- **Inheritance.** An other important property for the mutation function in genetic algorithm is inheritance. The problem-solving gene should be propagated to the next generation such that the whole genetic algorithm is approaching to the best solution.

Following these principles, we define the following enforcers for the PIs in this paper:

- **Off-road constraint enforcer.** Off-road constraint is the most frequent constraint in the planning invariant. If the violation of off-lane rule is generated by an initialization process, we directly move the position to the closest point where the whole object is off-road. We noticed that if a violation is introduced by a mutation, we can easily infer the direction of this violation by comparing the position of the points and the boundary of constraints. We still want to keep the direction of mutation due to the diversity. As shown in Fig. 11, suppose the object positions before and after the mutation are  $(s, l)$  and  $(s_m, l_m)$ , and we can infer that this mutation violates the constraints in the lateral direction with an intention to increase the  $l$ . The enforcer will thus apply the following transformation:

$$\begin{aligned} s_e &= s_m \\ l_e &= l_m + (\text{road\_width} + \text{obstacle\_width}) \end{aligned} \quad (8)$$

We keep the position on the direction where no violation happens and shift the position with the length of the constraint on the other direction. In this particular case, we keep the longitudinal direction of the object position unchanged since it does not violate the constraint. However since the lateral direction violates the constraint, the enforcer then apply additional offset to the lateral direction to make it satisfy the constraint. The offset we apply here is the width of the road plus the width of the AD vehicle, which is the whole width of the area where the static object can not be placed. By applying this offset, we transmit the object to another feasible area. As a result, we can maintain the randomness from the

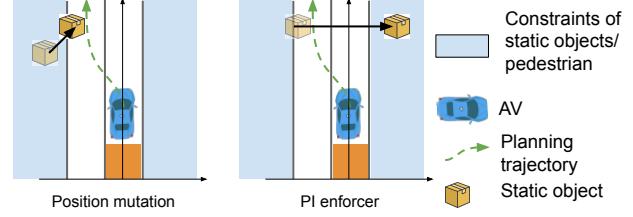


Figure 11: Off-road operation.

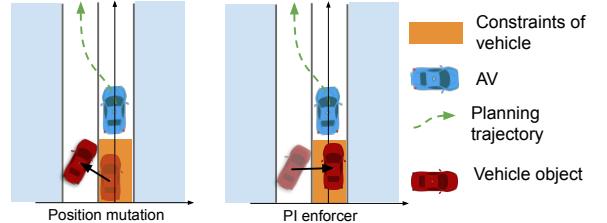


Figure 12: On-lane operation

mutation function and still enforce the constraints at the same time.

- **On-lane constraint enforcer.** On-lane constraint is used to force the vehicle driving in a normal way. Same as the off-road constraints handling approach, if the violation is generated during the initialization, we use the nearest lane way point to replace the original point. Following the same idea used in off-road constraints, we use the same transformation operation to enforce the constraints. As shown in Fig. 12, we still maintain the same mutation direction. When the boundary of the object has touched the constraint, we transmit the object to the other side of the lane. In the particular example in Fig. 12, we can notice that the position generated by the position mutation already touches the target lane of AD vehicle's lane changing, which violates the constraint of PI. During the enforcer step, we divide the leftwards lateral mutation into two parts: the first part drags the object to the boundary of the constraints. At this point, we initialize the position of object to the right side of the feasible area. By doing that, we can make sure that the leftwards position mutation will still mutate the object inside the constraints. Then we can start with the ramaining leftwards mutation.

## APPENDIX C

### DETAILS OF BP VULNERABILITY DISTANCE CALCULATION IN §IV-E

To calculate the BP vulnerability distance, we first need to apply the *attack target control* and *data dependency analysis* in the offline analysis and instrumentation phase. In this phase, we first build a Program Dependence Graph (PDG) to capture all the control and data dependence. Program dependence graph is a common way to capture the dependence relationships in the program [145, 146]. We followed the approach described in [146] to construct the PDG. First, we construct the intra-procedural PDG by combining the results of control

Table IV: Summary of Planning Invariants (PI) identified and used in the paper.

PI Index	Planning Scenario	Object Type	Constraints on Physical Objects	Desired Planning Behavior
PI1	Lane following (single-lane road)	Static obstacles	<b>PI-C1.</b> Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on	Keep cruising in the current lane
		Vehicles	<b>PI-C2.</b> Follow the AD vehicle	
		Pedestrians	<b>PI-C3.</b> Drive on reverse lane <b>PI-C4+5.</b> Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on	
PI2	Lane following (multiple-lane road)	Static obstacles	<b>PI-C1.</b> Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on	Keep cruising in the current lane
		Vehicles	<b>PI-C2.</b> Follow the AD vehicle <b>PI-C3.</b> Drive on other lanes	
		Pedestrians	<b>PI-C4+5.</b> Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on	
PI3	Lane change	Static obstacles	<b>PI-C1.</b> Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on	Finish changing to the targeted lane
		Vehicles	<b>PI-C2.</b> Follow the AD vehicle	
		Pedestrians	<b>PI-C3.</b> Drive on other lanes except current and targeted lanes <b>PI-C4+5.</b> Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on	
PI4	Lane borrow (due to a blocking obstacle)	Static obstacles	<b>PI-C1.</b> Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on	Finish borrowing the reverse lane and pass blocking vehicle
		Vehicles	<b>SP-PI-C1.</b> On-lane and in front of the blocking obstacle <b>PI-C2.</b> Follow the AD vehicle	
		Pedestrians	<b>PI-C3.</b> Drive on other lanes except current and targeted lanes <b>SP-PI-C2.</b> On-lane and park in front of the blocking obstacle <b>PI-C4+5.</b> Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on	
PI5	Intersection w/ stop sign	Static obstacles	<b>PI-C1.</b> 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	Pass intersection w/ stop sign following the traffic rule
		Vehicles	<b>PI-C2.</b> Follow the AD vehicle	
		Pedestrians	<b>PI-C3.</b> Drive on other lanes except current and targeted lanes <b>PI-C4+5.</b> Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on	
PI6	Intersection w/ traffic signal	Static obstacles	<b>PI-C1.</b> 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	Pass intersection w/ traffic signal following the traffic rule
		Vehicles	<b>PI-C2.</b> Follow the AD vehicle	
		Pedestrians	<b>PI-C3.</b> Drive on other lanes except current and targeted lanes <b>PI-C4+5.</b> Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on	
PI7	Bare intersection	Static obstacles	<b>PI-C1.</b> 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	Pass the bare intersection
		Vehicles	<b>PI-C2.</b> Follow the AD vehicle	
		Pedestrians	<b>PI-C3.</b> Drive on other lanes except current and targeted lanes <b>PI-C4+5.</b> Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on	
PI8	Parking	Static obstacles	<b>SP-PI-C3.</b> Placed on other parking spots	Park into an empty targeted parking spot
		Vehicles	<b>SP-PI-C4.</b> Parked on other parking spots	
		Pedestrians	<b>SP-PI-C5.</b> Walking pedestrians moving away from AD vehicle	

dependence, def-use dependence, and flow-insensitive read-after-write data dependence. We also follow their parameter tree approach to avoid whole-program-analysis for inter-procedural data dependence analysis. After getting the PDG, we traverse the dependence graph from the target position to extract all the predicates which will affect the final decision via control dependence or data dependence. We further divide the predicates into two parts: critical predicates and non-critical predicates. Critical predicates refer to the predicates which 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. The generated PDG of the example code (§III-A) is shown in Figure 13. In the figure, we can clearly see that predicate on line 22 is a critical predicate since it is connected with the attack target with one control dependence edge. The other predicate on line 17 is a non-critical predicate since the data dependence edge is required to connect with the attack target in PDG. Here we define the control flow distance for each predicate to measure the distance between a certain predicate and the target position. For the dependence chain between a predicate and a certain target position, the control flow distance (CFD) is defined as

$$CFD(p, T) = \min_{T_i \in T} \sum_{c \in Dep(p, T_i)} (CFD(c)) \quad (9)$$

Table V: The 40 collected BP input traces from LGSVL simulator for our evaluation (§V). *Lexus* and *Lincoln* are short for Lexus RX Hybrid 2016 and Lincoln MKZ 2017. More details (e.g., pictures) of the map, vehicle, and driving behaviors in each trace are on our website [34].

Scenario	Driving Behavior	Map	Vehicle	Duration (# of Planing Decisions)
Lane Follow (Single lane road)	Follow a 1-lane straight narrow road (2.7m lane width)	Single-lane road	Apollo: Lincoln Autoware: Lexus	15.0s (133) 25.4s (2394)
	Follow a 1-lane straight medium road (3.0m lane width)	Single-lane road	Apollo: Lincoln Autoware: Lexus	14.3s (121) 23.8s (2241)
	Follow a 1-lane straight wide road (3.5m lane width)	Single-lane road	Apollo: Lincoln Autoware: Lexus	18.6s (157) 24.6s (2037)
	Follow a 1-lane left-curved road	CubeTown	Apollo: Lincoln Autoware: Lexus	21.3s (209) 18.3s (1749)
	Follow a 1-lane right-curved road	CubeTown	Apollo: Lincoln Autoware: Lexus	17.6s (172) 21.3s (1978)
Lane Follow (Multiple lane road)	Follow a 2-lane straight road	San Francisco	Apollo: Lincoln Autoware: Lexus	18.7s (177) 15.4s (1379)
	Follow a 3-lane straight road	Modern City	Apollo: Lincoln Autoware: Lexus	14.3s (121) 21.3s (1840)
	Follow a 4-lane left-curved road	San Francisco	Apollo: Lincoln Autoware: Lexus	18.7s (181) 19.8s (1679)
	Follow a 4-lane right-curved road	San Francisco	Apollo: Lincoln Autoware: Lexus	21.5s (208) 25.9s (2379)
	Follow a 4-lane straight road	San Francisco	Apollo: Lincoln Autoware: Lexus	13.4s (129) 19.5s (1437)
Lane Change	Right change on a straight road	San Francisco	Apollo: Lincoln	21.2s (203)
	Left change on a straight road	San Francisco	Apollo: Lincoln	15.7s (138)
	Left change on a left-curved road	San Francisco	Apollo: Lincoln	13.4s (130)
	Right change on a left-curved road	San Francisco	Apollo: Lincoln	18.7s (172)
	Left change on a right-curved road	San Francisco	Apollo: Lincoln	16.4s (159)
Lane Borrow	Borrow lane on a straight narrow road (2.7m lane width)	Single-lane road	Apollo: Lincoln	25.9s (238)
	Borrow lane on a straight medium road (3.0m lane width)	Single-lane road	Apollo: Lincoln	28.7s (279)
	Borrow lane on a straight wide road (3.5m lane width)	Single-lane road	Apollo: Lincoln	30.5s (317)
	Borrow lane on a left-curved road	CubeTown	Apollo: Lincoln	27.3s (262)
	Borrow lane on a right-curved road	CubeTown	Apollo: Lincoln	33.2s (329)
Traffic Signal Intersection	Turn left at a 4-way intersection	San Francisco	Apollo: Lincoln	47.1s (453)
	Turn right at a 4-way intersection	San Francisco	Apollo: Lincoln	36.8s (329)
	Go straight at a 4-way intersection	San Francisco	Apollo: Lincoln	27.9s (288)
	Turn right at a 3-way intersection	San Francisco	Apollo: Lincoln	26.4s (233)
	Go straight at a 3-way intersection	San Francisco	Apollo: Lincoln	31.9s (308)
Stop sign Intersection	Turn left at a 4-way intersection	Shalun	Apollo: Lincoln	32.3s (334)
	Turn right at a 4-way intersection	Shalun	Apollo: Lincoln	27.9s (255)
	Go straight at a 4-way intersection	Shalun	Apollo: Lincoln	23.8s (220)
	Turn right at a 3-way intersection	Shalun	Apollo: Lincoln	33.2s (329)
	Go straight at a 3-way intersection	Shalun	Apollo: Lincoln	29.7s (283)
Bare Intersection Intersection	Turn left at a 4-way intersection	GoMentum Station	Apollo: Lincoln	37.9s (361)
	Turn right at a 4-way intersection	GoMentum Station	Apollo: Lincoln	42.3s (391)
	Go straight at a 4-way intersection	GoMentum Station	Apollo: Lincoln	30.1s (287)
	Turn right at a 3-way intersection	GoMentum Station	Apollo: Lincoln	29.2s (288)
	Go straight at a 3-way intersection	GoMentum Station	Apollo: Lincoln	38.5s (379)
Parking	Park to a front parking spot	GoMentum Station	Apollo: Lincoln	23.4s (228)
	Park to a left close parking spot	GoMentum Station	Apollo: Lincoln	30.5s (309)
	Park to a right close parking spot	GoMentum Station	Apollo: Lincoln	27.6s (263)
	Park to a left far parking spot	GoMentum Station	Apollo: Lincoln	24.3s (231)
	Park to a right far parking spot	GoMentum Station	Apollo: Lincoln	17.9s (163)

Table VI: The physical-property initialization and mutation ranges for the attack-introduced physical road objects used in our experiments (§V).

Input	Parameters	Description	Valid ranges	
Type	type	An enumeration that represents type of object	{Pedestrian, Vehicle, Bicycle, Static object}	
Position	x, y	Coordinate of the object center in 2D space	[ego.position.x - 80, ego.position.x + 80], [ego.position.y - 80, ego.position.y + 80]	
Size	length (m), width (m), height (m)	Length, width, height of the object bounding box	Pedestrian	0.50, 0.50, 1.80
			Vehicle	4.70, 2.06, 2.05
			Parked Bicycle	1.80, 0.50, 1.00
			Static object	(0.5, 2.0), (0.5, 2.0), (0.5, 2.0)
Velocity	velocity (m/s)	The absolute velocity. It will be transformed into a vector based on the heading when feeding into planning.	Pedestrian	(0, 1.4)
			Vehicle	Ego velocity
			Parked Bicycle	0
			Static object	0
Heading	heading (angle)	The heading of the physical object.	Pedestrian	Parallel to the current road or leave the road vertically
			Vehicle	Current road heading
			Parked Bicycle	[0, π)
			Static object	[0, π)
Trajectory	a list of waypoints	A list of the waypoints which represents the predicted trajectory of each object.	Pedestrian	Sampling on a straight line based on the heading
			Vehicle	Sampling based on road network
			Parked Bicycle	Empty
			Static object	Empty

where  $p$  is a predicate,  $T$  is the set of the target positions.  $Dep(p, T_i)$  is the dependence chain between predicate  $p$  and the target position  $T_i$  and  $c$  is a certain control dependence edge in the dependence chain between  $p$  and  $T_i$ .  $CFD(c)$  will calculate distance on control flow between two nodes of this control dependence edge. The control flow distance is purely calculated in the offline phase.

The overall BP vulnerability distance is the sum of the vulnerability distance of each single predicate. For the critical predicates, our distance metric want to guide the execution flow to the branch which is reachable from the attack target or the closer branch if two branches are reachable. So besides the static control flow distance, we also need to get runtime information to calculate the data flow distance for each single predicate. Considering the PDG in Figure 13, the predicate on line 13 is directly connected with the attack target position with a control dependence edge. The CFD of this predicate is 1. The CFD of predicate on line 7 is 2 since the path from the predicate to target position contains 2 control dependence edges and 1 data dependence edge.

For the critical predicates, we can directly infer which branch is reachable to the target positions or closer to the target positions. We record the execution number of each branch as  $n_1$  and  $n_2$ .  $n_1$  is the number of reachable branches and  $n_2$  is the execution number of the unreachable or farther branch. The data flow distance can be calculated as follow:

$$DFD(p) = \begin{cases} 1 & p \text{ not executed} \\ \frac{n_2}{n_1+n_2} \frac{Min_{op}}{Max_{history}} & \text{otherwise} \end{cases} \quad (10)$$

where  $Max_{history}$  is the maximal value of difference between two operands of this predicate in the evolutionary testing history and  $Min_{op}$  is the minimal difference between two operands in this specific execution. We want to use this term to measure the distance between two operands in this predicate such that the fitness function can be smaller when the operands

are approaching and finally change the result of this predicate towards a closer execution trace.

For the non-critical predicates, since we do not know which branch is the closer branch towards the target positions, we simply minimize the difference between two operands to get more diverse execution traces.

$$DFD(p) = \begin{cases} 1 & p \text{ not executed} \\ \frac{n_1}{n_1+n_2} \frac{Min_{op}^1}{Max_{history}^1} + \frac{n_2}{n_1+n_2} \frac{Min_{op}^2}{Max_{history}^2} & \text{otherwise} \end{cases} \quad (11)$$

In the above equation, we use number 1 and 2 to refer to different branches. Finally, the overall BP vulnerability distance is defined as

$$BPVD(T) = \sum_{p \in P_{crit}} CFD(p, T) DFD(p, T) + c \sum_{p \in P_{non-crit}} CFD(p, T) DFD(p, T) \quad (12)$$

where  $P_{crit}$  and  $P_{non-crit}$  is the set of critical predicates and non-critical predicates.  $c$  is a constant number to give smaller weight to the non-critical predicates since we cannot get the precise information of which branch is closer to the target positions for the non-critical predicates.

We use the motivating example in §III-A to demonstrate the design of BP vulnerability distance. As shown in Figure 6, imaging that the static obstacles are placed increasingly closer to the victim AD vehicle’s driving lane until the DoS vulnerability is triggered. At the starting point, no obstacle is around the driving lane. Then the lateral distance between two static obstacles will become closer until the DoS vulnerability is triggered. If we directly use the output of the planner as the fitness value, we can only get a sharp drop when the DoS vulnerability is triggered, since before that the planning behavior is unchanged. If we applied the idea of directed greybox fuzzing [92] and use their defined distance, we can only get two sharp drops since that is the only changes of the control flow graph. However, benefit from the fine-grained

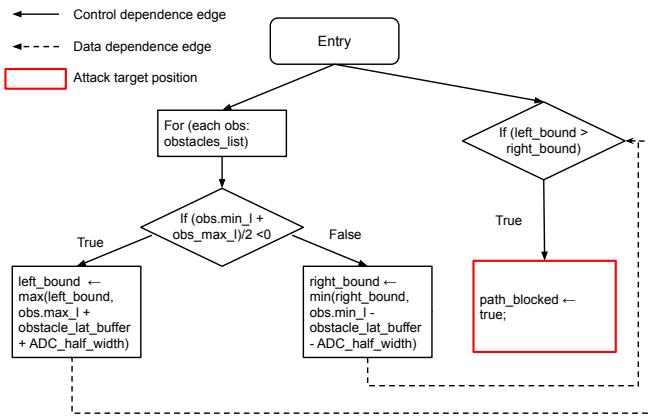


Figure 13: PDG graph of code example.

control and data dependence distance metrics, our BP vulnerability distance design can successfully capture the approaching tendency of the static obstacles during the mutation, since the distance of the predicates on line 7 and 13 is getting smaller. At the beginning, since there is no obstacles around, both predicates are not executed and the DFD of two predicates are 1. The BP vulnerability distance is  $1+2c$ . When the static obstacles are placed closer, both predicates are executed. The DFD of predicate on line 7 is getting smaller since the distance between obstacles and the center line is smaller. The DFD of predicate on line 13 is getting smaller since the distance between two obstacles is smaller. Once the DoS vulnerability is triggered, the DFD of the predicate 13 becomes 0 since the correct branch is executed.

#### APPENDIX D

#### DETAILS OF EACH DISCOVERED VULNERABILITY

##### A. V1. Apollo Lane Following

We have described the decision logic, the discovered vulnerability, and exploits in details in §III-A.

##### B. V2. Apollo Lane Changing

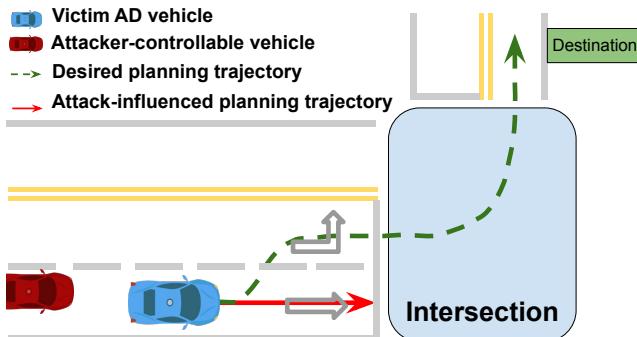


Figure 14: Attack scenario setup: Block the AD vehicle from changing lane with a following vehicle.

**Vulnerable code.** We presented the pseudo code of the vulnerability in Fig. 10. The whole program shown in the

figure is used to decide whether it is safe to perform lane changing given the positions of surrounding vehicles. All the position variables in the code are in the Frenet coordinate based on the target lane of the lane changing. The first check on line 5 is used to check whether a surrounding vehicle is located on the target lane. If not, this vehicle will be ignored. The second check on line 10 is used to check the longitudinal distance between the AD vehicle and the vehicle. If the distance is smaller than a backward safety distance, then Apollo will come to the conclusion that it is not a good time to change the lane since another vehicle is driving on the target lane and the longitudinal distance between them is less than a safety distance. The vulnerable code is on line 5. Since the lateral buffer is too large, a vehicle following the AD vehicle is also considered as a blocking vehicle (usually the half lane width is smaller than 2m even on the highway [69]).

**Attack scenario setup.** As shown in Fig. 14, we design an attack scenario where the attacker can force the victim AD vehicle to give up lane changing during the whole driving by exploiting V2. The victim AD vehicle needs to change the lane so that it can turn left at the intersection to reach to the destination. The attacker just needs to drive a car following the victim AD vehicle so that the check on line 5 can not be satisfied. As a result, the victim AD vehicle always considers the attacker’s vehicle as a blocking vehicle and it is not safe to change the lane. In the end, the victim AD vehicle has to re-route to get to the destination or stop in the current lane to wait until it thinks it is clear to change the lane. If the AD vehicle has to re-route, the overall travelling time and energy consumption will greatly increase and damage the passengers’ experience. Stopping to wait for lane changing will also block the normal traffic operations and is considered traffic violation in some cases [74]. As a result, both behaviors will increase the traffic load, which contradicts with one of the benefits of autonomous driving: efficiency and convenience, based on a report from United States department of transportation [147].

**End-to-end result.** We construct the attack scenario in LGSVL [98]. The whole scenario happens in front of an intersection where the victim AD vehicle has to turn left to get to the destination. By carefully controlling the trajectory of a attacker-controllable vehicle with the API provided by LGSVL, we are able to force the victim AD vehicle to stay on the right lane while our controlled vehicle is always driving on the right lane. In benign scenario (left figure in Fig. 9), we control an vehicle following the AD vehicle and our controlled car is driving slightly on the right side of the lane (to make sure the vehicle is skipped by line 5). Apollo does not consider the vehicle as blocking the road and changes the lane successfully. For the scenario under attack (right figure in Fig. 9), we controlled the vehicle to follow the car and drive on the left side of the lane to pass the lateral position check on line 5. As a result, our controlled car can block the victim AD vehicle so that it fails to change the lane until it reaches the intersection.

##### C. V3. Apollo Lane Borrow (In-lane Static Object)

```

1 # Iterate over the obstacle list
2 for (obs : obstacle_list):
3     if (obs.id == current_obs.id):
4         continue
5     #Check lateral intersection
6     if (obs.start_l > current_obs.end_l or
7         obs.end_l < current_obs.start_l):
8         # Skip if no lateral intersection
9         continue
10    delta_s ← obs.start_s - current_obs.end_s
11    if(delta_s<0 or delta_s> Threshold)
12        continue
13    # Make decision that current_obs is temporarily
14    # blocked by another obstacle and will move
15    # forward once its blocking obstacle is cleared
16    return false
17 # Make decision that current_obs is non-movable
18 return true

```

Figure 15: Simplified vulnerable pseudo code of V3 (lane borrow + on-lane objects) in Apollo. This is part of a function which decides whether the blocking obstacle is movable or not.

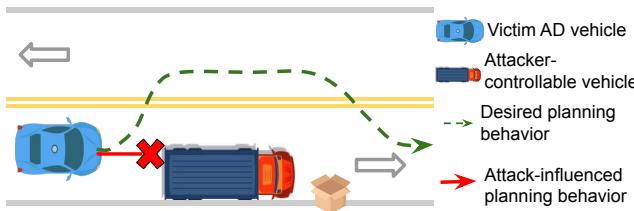


Figure 16: Attack scenario setup: A static box blocks the victim AD vehicle under a lane borrow scenario.

**Decision logic.** This part is executed to make decision for lane borrow behavior, which happens in a single-lane road scenario and there is a blocking vehicle or other object blocks the current lane. As a result, the AD vehicle has to borrow the reverse lane to bypass the blocking object. The decision logic shown in Fig. 15 is designed to recognize the pattern of traffic queue caused by checking whether the front vehicle is blocked by another one. If so, the decision logic thinks the front vehicle is temporarily blocked by another one and will move forward later. Since there exists a queue on the current lane and the AD vehicle should wait on the current lane. This is a very common scenario, for example, a line of vehicles wait statically due to a traffic congestion. In this case, the AD vehicle should not borrow the reverse lane since, which will potentially make the traffic congestion even worse. This part of decision logic is necessary since the lane borrow behavior is designed for non-movable obstacles which can be passed by borrowing the reverse lane, which should not be applied when there is a traffic queue.

**BP DoS vulnerability.** Our testing system is able to discover a vulnerability in the aforementioned decision logic. The triggering condition is that if we put another static physical object in front of the blocking vehicle and with lateral intersection, the decision logic will mistakenly think there exists a traffic queue and the blocking vehicle is waiting for the newly-added object. As a result, the AD vehicle should wait statically until others objects to move. However, all of the blocking objects are static and they will not move. This

will directly cause permanent stop of the AD vehicle.

**Potential exploitation.** To exploit this vulnerability in real world, the attacker has to first create a lane borrow scenario. This requires a single-lane road and a blocking object on the lane. Since the current lane is blocked by the attacker, so the victim AD vehicle has no choice except borrowing the reverse lane. The attacker can control a vehicle parking on the lane with the emergence light on. This is very common in the real world, for example, a taxi is waiting for its passenger and a truck is unloading on the roadside. The attacker also has to control a static object in front of the blocking vehicle with lateral intersection to trigger the vulnerable logic and make the blocking vehicle movable. As a result, the victim AD vehicle will wait for the blocking vehicle to move and thus permanently stop. Note that there is no specific requirement for the second object, the only requirement is that it has to have lateral intersection with the blocking vehicle and has to be static. In the piratical attack scenario we create in Fig. 16, the attacker can park the blocking vehicle slightly right so that a small part of the vehicle is off-road. Then the attacker can put a static object, such as an easy-to-carry cardboard box purely off-road to change the decision of the AD vehicle and make it stop permanently.

#### D. V4. Apollo Lane Borrow (Off-lane Static Obstacles)

Fig. 17 shows the simplified pseudo code for a BP DoS vulnerability our system discovered from the 5.0 version of Apollo.

**Decision logic.** As shown in Fig. 17, the overall goal of this code snippet is to calculate the blocked perception angle by each obstacles in front of the AD vehicle and determine whether the perception is blocked by an obstacle. Specifically, it maintains two variables `left_most_angle` and `right_most_angle`, which are resolved by beam scanning to calculate the angle range of blocked perception for a particular obstacle. The decision logic will uniformly sample the beam angle in front of the AD vehicle and check whether the obstacle has intersection with the beam on that direction. Once the beam first touches the obstacle, the `right_most_angle` will be updated (line 25). And when the beam no longer touches the obstacle, the `left_most_angle` will be updated (line 28). If the angle blocked by a certain obstacle is larger than the threshold `block_angle_threshold`, then the decision logic will come to the conclusion that the current perception is blocked by this obstacle. As a result, the AD vehicle will stop during the lane borrow since it may be dangerous to take an aggressive driving behavior (borrowing the reverse lane) when the perception is blocked.

**BP DoS vulnerability.** Our testing system is able to discover a DoS vulnerability in the aforementioned decision logic. Fig. 18 provides a malicious physical objects setup to trigger this vulnerability. The code snippet first performs the beam scanning for the right cardboard box and updates the two angle boundaries since the right box is in front of the AD vehicle. Note that the flag `right_most_found` is set to be true. While the code

```

1 #scanning range centering at AD vehicle heading
2 search_range ← π
3 #length of scanning beam in meter
4 search_beam_length ← 20
5 #resolution of the beam scanning
6 search_beam_intensity ← 0.08
7 #checking if perception is blocked by obstacle
8 block_angle_threshold ← 0.5
9 left_most_angle ← Normalize(adv_heading + 0.5 *
    → search_range)
10 right_most_angle ← Normalize(adv_heading - 0.5 *
    → search_range)
11 right_most_found ← false
12
13 #Iterate over the obstacle list
14 for (obs : obstacle_list):
15     # Start beam scanning
16     for (search_angle=0; search_angle<search_range;
17         search_angle+=search_beam_intensity):
18         #Calc beam heading based on AD veh heading
19         beam_heading ← adv_heading - 0.5 *
            → search_range + search_angle
20         #Check if obstacle has intersection with beam
21         overlap ← obs.Overlap(adv_pos, beam_heading,
            → beam_length)
22         #Update right_most_angle when beam first
            → encounter obs
23         if (not(right_most_found) and overlap):
24             right_most_found ← true
25             right_most_angle ← beam_heading
26         #Update left_most_angle if no longer overlap
            → with obs
27         if (right_most_found and not(overlap)):
28             left_most_angle ← beam_heading -
                → search_angle
29             break
30         if (not(right_most_found)):
31             continue
32         # Compare angle range with threshold to make
            → decision
33         if (abs(Normalize(left_most_angle -
            → right_most_angle))>block_angle_threshold):
34             # Make decision whether the perception is
            → blocked
35     return true
36 # false means perception is blocked by an obstacle
37 return false

```

Figure 17: Simplified vulnerable pseudo code of V4 (lane borrow + off-road objects) in Apollo. This is part of a function which estimates if an obstacle in front of AD vehicle blocks too much space of perception by beam scanning.

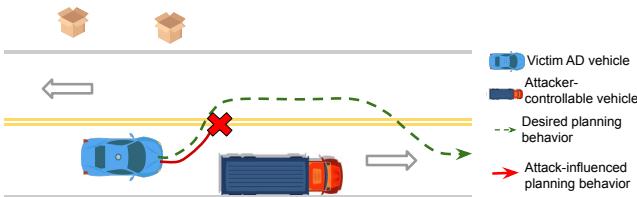


Figure 18: Attack scenario setup: Two static off-road boxes block the victim AD vehicle under a lane borrow scenario.

```

1 # Blocking condition is set as AD vehicle's width
2 threshold ← adc_width
3 # Check if the obs is static
4 if (obs.is_static or obs.trajectory.empty):
5     # Checks relationship between driving path & obs
6     if(reference_line.IsBlock(obs.perception_box,
        → threshold)):
7         #Make decision that obs crosses driving path
8         obs.is_path_cross ← true

```

Figure 19: Simplified vulnerable pseudo code of V5 (lane borrow + standing pedestrian) in Apollo. This is part of the code used to check whether the AD vehicle should stop in front of a crosswalk due to the pedestrian.

performs the beam scanning for the left cardboard box, line 28 will be executed immediately since `right_most_found` is true. As a result, the `left_most_angle` will be set to be `adv_heading - 0.5 * search_range`. When the decision is made on line 33, the difference between `left_most_angle` and `right_most_angle` will be large since one is on the most left side, while the other one is on the most right side. As a result, the decision logic make the decision that perception is blocked due to a static object even not in the search range.

The root cause behind this is the implementation error. We notice that in the code comment, the Apollo's developer clearly describe the function of this part of decision logic [148] is to estimate if an obstacle in certain range in front of ADV blocks too much space perception behind itself. If the decision is made separately for each obstacle, then clearly the variable `left_most_angle`, `right_most_angle`, `right_most_found` should updated for each obstacle.

**Potential exploitation.** To exploit this vulnerability in real world, the attacker can prepare two easy-to-carry static objects such as cardboard boxes, which are not too uncommon in road regions. The attacker needs to put the two objects in the off-road area and visible to the LiDAR and camera sensors of the AD vehicle.

#### E. V5. Apollo Intersection with Traffic Signal (Standing Pedestrian)

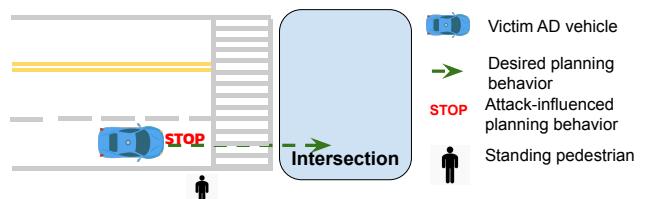


Figure 20: Attack scenario setup: An off-road standing pedestrian force the victim AD vehicle to stop in front of the crosswalk.

**Decision logic.** As shown in Fig. 19, the decision logic determines whether a certain obstacle overlaps with the driving path (or the driving reference line) on a crosswalk. Note that the AD vehicle will stop in front of the crosswalk if the `is_path_cross` flag is true for any pedestrian. The decision logic shown in Fig. 19 is designed specific for the static obstacles. Thus, line 4 skips all the dynamic obstacles. Line

```

1 # A lateral distance configuration
2 strict_l_distance ← 6 # unit: meter
3 # Compare lateral distance of obstacle with
   → configured value
4 if (l_distance < strict_l_distance):
5   # Get vector between positions of AD vehicle and
   → obstacle
6   obs_to_adc ← adc_path_point - obstacle.position
7   # Compare inner product with a pre-defined
   → threshold
8   if (InnerProduct(obs_to_adc.v, obs_to_adc) >
   → 1e-6):
9     # Make decision that pedestrian is moving
   → towards AD veh
10    stop ← true

```

Figure 21: Simplified vulnerable pseudo code of V6 (intersection + walking pedestrian) in Apollo. This is part of the code used to check whether the AD vehicle should stop in front of a crosswalk due to the pedestrian.

6 calls the function `IsBlock` to determine the relationship between perception box of the obstacle and driving reference line. This function takes two arguments: the obstacle bounding box and a threshold. It will calculate the minimum distance between reference line and the bounding box and returns true if the minimum distance is smaller than the threshold.

**BP DoS vulnerability.** Our system discover a DoS vulnerability caused by a standing off-road pedestrian. The reason that the off-road pedestrian is that the threshold set on line 6 is overly-conservative such that a pedestrian off-road is still within the range.

**Potential exploitation.** To exploit this vulnerability, the attacker needs to control a pedestrian which keep standing near a crosswalk. As a result, the victim AD vehicle will got stuck in front of the crosswalk. Note that crosswalk is very common in intersection, the victim AD vehicle could greatly block the normal traffic and violate the traffic rule [75] if exploited at an intersection.

#### F. V6. Apollo Intersection with Traffic Signal (Walking Pedestrian)

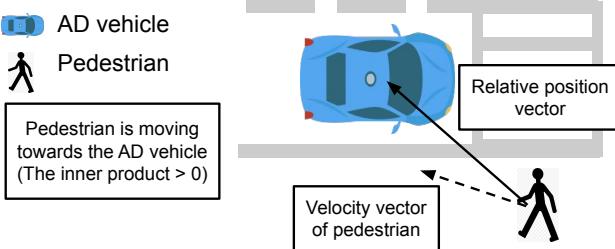


Figure 22: An explanation of how decision logic determines the pedestrian is moving towards the AD vehicle

**Decision logic.** As shown in Fig. 21, the AD vehicle will stop in front of the crosswalk if two requirements are satisfied: (1) the lateral distance between the obstacle and the driving reference line is within a predefined range, and (2) the pedestrian is moving towards the AD vehicle. Once the lateral distance requirement is satisfied, the decision logic calculates the inner product between the velocity of the pedestrian and

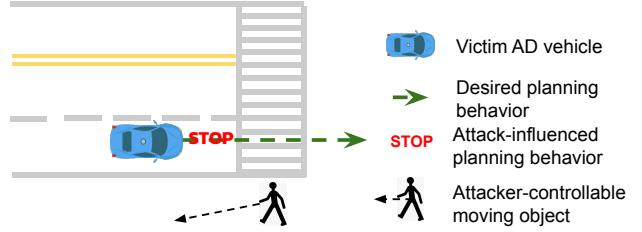


Figure 23: Attack scenario setup: An off-road moving pedestrian force the victim AD vehicle to stop in front of the crosswalk.

its relative position from the AD vehicle to determine whether the pedestrian has a moving tendency towards the AD vehicle, as illustrated in Fig. 22. The principle behind this is that when the inner product calculated on line 8 is positive, the angle between the velocity and the relative position is an acute angle and when the pedestrian moves along its current velocity direction, the distance between the AD vehicle and the pedestrian will become closer. Thus, in this case, the AD vehicle decides to stop to yield the pedestrian.

**BP DoS vulnerability.** Our system discovers a semantic DoS vulnerability in the above decision logic. When the AD vehicle is approaching to a crosswalk, a pedestrian who just passes the crosswalk and leaves the intersection along the road will make the victim AD vehicle to stop since the angle between the velocity and relative position is an acute angle.

**Potential exploitation.** To exploit this vulnerability in the real world, as shown in Fig. 23, the attacker could keep a moving angle such that she is leaving the intersection while keeping an acute angle between vector vector and relative position vector. As a result, the AD vehicle will always make decision to stop in front of the crosswalk during this process. Also, the change of the angle is a dynamic process, the attacker may carefully choose a moment to change the angle to acute such that there is enough space for the victim AD vehicle to decelerate. The victim AD vehicle has to brake with the maximal deceleration, which will greatly damage the passengers' experience and potentially cause traffic accident, such as rear-end collision. Moreover, due to the overly conservative logic in Apollo, it will treat an unknown moving object as a pedestrian. A more interesting attack scenario is that the attacker may use multiple rovers, which can not be recognized by the current AD system and classified into unknown type to exploit this vulnerability. The attacker can carefully design the moving trajectory of the rovers and make sure there is always one rover can exploit this vulnerability. As a result, the victim AD vehicle will stop permanently.

#### G. V7. Apollo Intersection with Stop Sign

**Vulnerable code.** We presented the vulnerable code logic in Fig. 24. Due to the traffic rule that a vehicle must wait until all the vehicles arrive at the intersection earlier have left, Apollo has to maintain a watch list. When the AD vehicle is approaching to an intersection with stop sign, an object will be put into the watch list if it is a dynamic object (line 4) and it

```

1 # Iterate over all the obstacles
2 for (each obs : obstacle_list):
3     # skip all the static objects
4     if (obs.type!=BICYCLE and obs.type!=VEHICLE):
5         continue
6     # Get the closest lane within 5m.
7     obstacle_lane ← GetClosestLane(obs.position,
8         ↪ 5.0)
9     # Check if obstacle_lane is an associated
10    # lane guarded by a stop sign
11    if(obstacle_lane != NULL and obstacle_lane in
12        ↪ asso_lanes):
13        if (obs.distance_to_stop_line <
14            ↪ max_stop_distance):
15            #Add obstacle into watch list AD veh
16            #has to wait until watch list is empty
17            watch_list.add(obstacle_lane)

```

Figure 24: Simplified vulnerable pseudo code of V7 (stop sign) in Apollo.

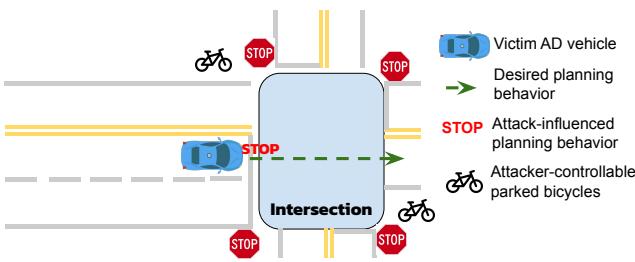


Figure 25: Attack scenario setup: Force the AD vehicle to permanently stop with 2 off-road static bicycles.

is located on a lane guarded by a stop sign (line 10). Also, the object needs to stop in front of the stop line (line 11). After satisfying all the conditions above, that object will be added into the watch list. The problem is that on line 4, Apollo does not skip a static bicycle and on line 7, the distance buffer is too large such that off-road objects are also considered to be on the lane. As a result, a static bicycle without any cyclist off the road will be mistakenly put into the watch list. Apollo BP in fact has a timeout design, where the AD vehicle will still move forward when the watch list contains only one vehicle or bicycle and the timeout duration has passed. However, such design can be easily bypassed if the attacker puts two static bicycles off the road.

**Attack scenario setup.** As shown in Fig. 25, we design an attack scenario with a 4-way stop sign intersection. The attacker needs to put two static bicycles within 5m of an associated lane guarded by the stop sign. In our setup, we put two static bicycles to bypass the timeout design. As a result, the victim AD vehicle will stop in front of the stop line permanently.

#### H. V8. Autoware Lane Following (Static Objects)

**Vulnerable Code.** We presented the pseudo code of the vulnerable planner logic in Autoware in Fig. 26. This part of code is designed to handle static objects. Autoware generates a list of parallel candidate planning trajectories for the AD vehicle and check whether each trajectory is blocked by the static obstacle one by one. Specifically, Line 9 iterates all the contour points of a physical object to get precise

```

1 # Lateral threshold to decide the blocking object
2 critical_lateral_distance ← car_width/2 + 1.2f
3 # Longitudinal distance threshold to ignore a veh
4 minFollowingDistance ← 35.0f
5 for (each obs : obstacle_list)
6     # Iterate all the candidate trajectories
7     for (each traj : candidate_trajectories):
8         # Iterate all the contour points of an object
9         for (each point : obs.contourPoints):
10            # Calculate the lateral/longitudinal
11            ↪ distance
12            # between trajectory and contour point
13            lateraldist←LateralDistance(traj,point)
14            longitudinaldist ←
15            ↪ LongitudinalDistance(traj, point)
16            # Check whether the trajectory is blocked
17            if (lateraldist <= critical_lateral_distance
18                and longitudinaldist >= -car_length/1.5
19                and longitudinaldist <
20                ↪ minFollowingDistance):
21                traj.isBlocked ← true
22            # Boolean to indicate if AD veh is fully blocked
23            FullyBlocked ← true
24            for (each traj: candidate_trajectories):
25                if (traj.isBlocked == false):
26                    FullyBlocked ← false

```

Figure 26: Simplified vulnerable pseudo code of V8 (lane follow) in Autoware.

relationship between the candidate trajectory and the physical object. *lateraldist* and *longitudinaldist* are calculated based on the current candidate trajectory and represent the lateral and longitudinal distances between the AD vehicle center and the contour point. On line 15, if the lateral distance is within the critical lateral distance range and it is ahead of the AD vehicle within the following distance, the object is considered as a blocking static object on line 18. After that, it then determines whether the AD vehicle is fully blocked by merging the decisions on the candidate trajectories (line 20–23). A fully blocking decision is made if all candidate trajectories are blocked by the static object. We can exploit this logic by putting static objects off the road and block all the candidate trajectories since the *critical\_lateral\_distance* is much larger than the half lane width [69].

**Attack scenario setup** We create an attack scenario where the attacker put two cardboard boxes off the lane to block all the candidate trajectories of the victim AD vehicle. As a result, the victim AD vehicle stops in the middle of the lane permanently. The permanent stop block the normal traffic operations and may directly violate the traffic rules. The improper stop can further cause severe consequences if this vulnerability is triggered inside a tunnel or in front of a fire station [74, 75].

#### I. V9. Autoware Lane Following (Dynamic Objects)

**Decision logic.** As shown in Fig. 27, the decision logic is designed to check whether a candidate planning path will be blocked by a dynamic physical object with the prediction trajectory *pred\_traj*. Similar to the decision logic in Fig. 26, it makes decision whether the current lane is blocking by checking whether all the candidate planning trajectories are blocked. Specifically, on line 8, *collision\_distance*

```

1 #Pre-defined lateral distance threshold
2 critical_lateral_d=AD_half_width+1.2f #unit: meter
3 #Iterate over each point in predicted obj
   ↳ trajectory
4 for (pred_point : pred_traj):
5   #Iterate over each point in planning path of AD
   ↳ veh
6   for (path_point : path):
7     #Compute distance between two points
8     collision_distance ← distance(pred_point,
       ↳ path_point)
9     #Compare distance with pre-defined threshold
10    if (collision_distance < critical_distance):
11      #Make decision: candidate trajectory is
         ↳ blocked
12      blocked ← true

```

Figure 27: Simplified vulnerable pseudo code of V9 (lane following + dynamic vehicles) in Autoware. This is part of the code used to check whether current lane is blocked by dynamic objects.

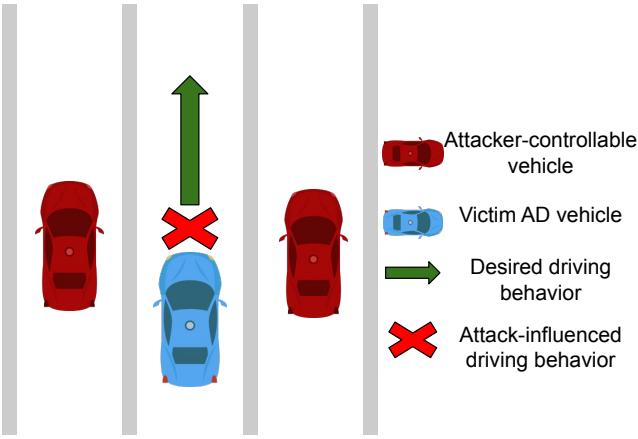


Figure 28: Attack scenario setup: Force the AD vehicle to sharp brake with 2 driving vehicles on adjacent lanes.

is calculated between each point on the prediction trajectory of the physical object and the point on planning path. Once this distance is smaller than the pre-defined threshold `critical_lateral_d`, the decision logic will consider this as a potential collision point and makes the decision that the planning path is blocked.

**BP DoS vulnerability.** Our system discover a DoS vulnerability caused by driving vehicles on adjacent lanes. The reason that vehicles on adjacent lanes could change the desired planning behavior into an overly-conservative decision is the large lateral distance threshold defined on line 2. Given that the common urban lane width is smaller 3.6m [69], the vehicles on adjacent lane can easily be within the range defined by the lateral threshold and thus affect the driving decision.

**Potential exploitation.** As shown in Fig. 28, a possible exploitation scenario is that the attacker controls two vehicles driving on the two adjacent lanes near the victim AD vehicle. The attacker can carefully choose the timing such that both the attacker-controllable vehicles are observed by the victim AD vehicle at the same time and within the blocking decision range. For example, two attacker-controllable vehicles can accelerate and appear in front of the victim AD vehicle at

the same time. As a result, the victim AD vehicle will come to the conclusion that all the candidate trajectories are blocked by the two vehicles and it should decelerate to avoid collision. This will lead to a sharp brake, which will greatly damage the passengers' experience and cause potential rear-end collision.