LawBreaker: An Approach for Specifying Traffic Laws and Fuzzing Autonomous Vehicles

Yang Sun
Singapore Management University
Singapore
yangsun.2020@phdcs.smu.edu.sg

Christopher M. Poskitt
Singapore Management University
Singapore
cposkitt@smu.edu.sg

Jun Sun
Singapore Management University
Singapore
junsun@smu.edu.sg

Yuqi Chen
ShanghaiTech University
China
chenyq@shanghaitech.edu.cn

Zijiang Yang
GuardStrike Inc.
China

ABSTRACT

Autonomous driving systems (ADSs) must be tested thoroughly before they can be deployed in autonomous vehicles. High-fidelity simulators allow them to be tested against diverse scenarios, including those that are difficult to recreate in real-world testing grounds. While previous approaches have shown that test cases can be generated automatically, they tend to focus on weak oracles (e.g. reaching the destination without collisions) without assessing whether the journey itself was undertaken safely and satisfied the law. In this work, we propose `LawBreaker`, an automated framework for testing ADSs against real-world traffic laws, which is designed to be compatible with different scenario description languages. `LawBreaker` provides a rich driver-oriented specification language for describing traffic laws, and a fuzzing engine that searches for different ways of violating them by maximising specification coverage. To evaluate our approach, we implemented it for Apollo+LGSVL and specified the traffic laws of China. `LawBreaker` was able to find 14 violations of these laws, including 173 test cases that caused accidents.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

KEYWORDS

Autonomous vehicles, traffic laws, fuzzing, STL, LGSVL, Apollo

ACM Reference Format:
Yang Sun, Christopher M. Poskitt, Jun Sun, Yuqi Chen, and Zijiang Yang. 2022. LawBreaker: An Approach for Specifying Traffic Laws and Fuzzing Autonomous Vehicles. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556897

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556897

1 INTRODUCTION

Autonomous driving systems (ADSs) combine sensors and software to control, navigate, and drive autonomous vehicles (AVs). As inherently safety-critical systems, ADSs must be comprehensively tested before they can be deployed on public roads. High-fidelity simulators (e.g. LGSVL [53], CARLA [17]) play an important role in this effort as they allow ADSs to be evaluated across a broad range of scenarios. This includes scenarios that are hard to recreate in real-world testing grounds, but are important to evaluate since an incorrect decision by the ADS could lead to an accident [16, 23].

In black box simulator-based testing, the ADS is systematically evaluated against a number of different *scenarios* and *oracles*. Scenarios are configurations of objects on a map (e.g. obstacles, pedestrians, and vehicles) as well as their dynamic behaviour, and can be described to different degrees by domain-specific languages (DSLs) such as Scenic [25], CommonRoad [7], GeoScenario [49], and AVUnit [3]. Oracles are 'pass/fail' criteria that the ADS must satisfy under every test scenario [45]. Unfortunately, existing testing frameworks tend to use weak oracles. AV-Fuzzer [37], for example, evaluates ADSs on their ability to complete a journey without getting too close to other vehicles. Criteria based on getting from A to B without collisions are no doubt important, but for AVs, the journey is as important as the destination, and we need richer criteria about how an AV undertakes it. Jumping red lights at every junction is clearly unacceptable, for example, even if the ADS manages to achieve it without collisions.

Fortunately, rich sets of criteria for how a vehicle should undertake a journey already exist: the various national *traffic laws*. In addition to avoiding collisions, an ADS should satisfy the traffic laws of the country it operates in. Until we design new traffic laws specifically for ADSs, existing traffic laws remain the gold standard for ensuring road safety. Testing an ADS against such traffic laws, however, is challenging. First, they are typically expressed in natural language with respect to the driver's perspective. This leads to non-intuitive encodings in existing specification languages that are based on a global view (e.g. [3]). Second, traffic laws vary across countries, so a general and adaptable specification language is necessary (instead of a fixed built-in oracle for one country). Unfortunately, existing specification approaches for traffic laws have limited reusability and extensibility. For example, rulebooks [11, 13] focus on the logic transition process and do not

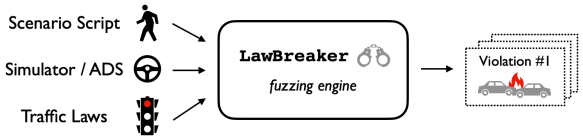

Figure 1: High-level workflow of `LawBreaker`

provide a natural way to describe laws, whereas other formalisations (e.g. [19, 39, 50, 51]) are tightly coupled with the test scenarios, i.e. the laws must be customised for each new scenario.

In this work, we present the design and implementation of `LawBreaker`, a DSL for specifying traffic laws and an automated framework for testing ADSs against them. First, our language allows users to specify traffic laws more naturally from the perspective of the driver (instead of globally), e.g. "at a junction the driver should give way to pedestrians when turning right". Second, `LawBreaker` is decoupled from any particular testing scenario, i.e. not only can the same laws be interpreted across different maps, but they can be used together with any DSL for generating test scenarios (i.e. placements of vehicles, pedestrians, and obstacles in a map). Finally, `LawBreaker` provides a fuzzing engine that searches for different violations of laws by attempting to cover as many different ways of violating the specification as possible. These uncovered violations may then provide clues on how to improve ADSs.

The workflow of `LawBreaker` is summarised in Figure 1. Users provide a scenario script, an ADS, a simulator, and some traffic laws specified using our language. Our fuzzing engine then systematically generates test cases in the simulator that try to cause the ADS to violate those laws, revealing flaws to be addressed in the design of the ADS. These violations are recorded, and can be played back visually by using the simulator.

Our implementation of `LawBreaker` consists of: (1) a grammar parser, which uses antlr4 [4], to extract the elements describing a scenario and the corresponding traffic laws; (2) a fuzzing engine, that implements our specification-coverage guided fuzzing algorithm; and (3) a bridge, which connects the grammar parser, fuzzing engine, ADSs, and simulator to make the whole system run. We evaluate this implementation using AVUnit [3] for scenario scripts, LGSVL [53] as the simulator, and different versions of Baidu Apollo [1, 2] as the ADSs under test. As Apollo was designed by a Chinese company, we chose to evaluate it under Chinese traffic laws. In particular, we specified and tested 24 Chinese traffic laws in `LawBreaker`, finding that 14 of them were violated by Apollo, and that 173 of the test cases generated also caused accidents. Videos of some of these violations can be found online [6].

2 OVERVIEW OF LAWBREAKER

The overall architecture of `LawBreaker` and how it interfaces with existing simulators and ADSs is shown in Figure 2. It has three main components: an existing DSL for describing scenes and scenarios (AVUnit [3]); our new driver-oriented specification language for traffic laws, based on signal temporal logic (Section 3); and the `LawBreaker` fuzzing algorithm (Section 4). Our architecture is fully decoupled, and intended to be compatible with different ADSs (e.g. different versions of Apollo [2] and Autoware [5]).

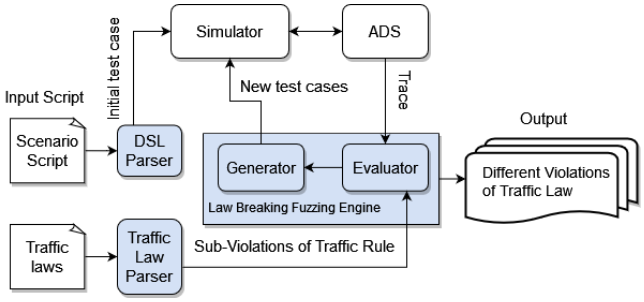

Figure 2: The architecture of `LawBreaker`

First, the scenario script component prepares the initial test case of the simulator by translating the specified scenario into the required API calls. Second, the traffic law component describes testing oracles from the driver's view without needing any particular knowledge about the map or its agents. Finally, the fuzzing engine repeatedly extracts a trace from the ADS, evaluates it against the specification, and uses the outcome to generate new test cases for the simulator to run. Note that this algorithm uses the grammar of the scenario DSL when generating new testing scenarios. The simulator itself (e.g. LGSVL or CARLA) treats the ADS as a black box. When tests are underway, the ADS extracts sensory information of the AV from the simulator. This includes data from perception equipment (e.g. camera, Lidar, and GPS) and chassis control (e.g. brake, gas, and steer).

In order to specify testing scenarios, we utilise an existing DSL called AVUnit [3]. AVUnit specifies the motions of NPC vehicles and pedestrians (i.e. non-player characters representing objects other than the AV under test), and other environment-related information such as time and weather. AVUnit is a highly expressive language which allows us to specify detailed scenarios, e.g. the status of every NPC vehicle as well as their trajectories. We refer the readers to [3] for details on AVUnit and remark that alternative scenario DSLs (e.g. Scenic [25]) can be adopted for `LawBreaker` easily. Note that even though AVUnit describes the motion task of the ego vehicle, the trajectory of the ego vehicle is determined by the ADS.

Illustrative Example. Listing 1 presents an example of a traffic law specification in `LawBreaker`. In particular, it describes Article #38 of the *Regulations for the Implementation of the Road Traffic Safety Law of the People's Republic of China* [12], which stipulates how a vehicle should behave with respect to a traffic light at a junction. An English translation of Article #38 reads as follows:

(1) When the green light is on, vehicles are allowed to pass, but turning vehicles shall not hinder the passing of vehicles going straight and pedestrians who are crossing;
(2) When the yellow light is on, vehicles that have crossed the stop line can continue to pass;
(3) When the red light is on, vehicles are prohibited from passing. However, vehicles turning right can pass without hindering the passage of vehicles or pedestrians.

Article #38 is specified as `law38` in Listing 1, which in turn consists of three conjuncts separately describing clauses (1)–(3). In this example, we focus on the rules concerning yellow lights in (2), which are specified as `law38_sub2` (Line 10). Note that `law38_sub2` is a

```
Trace trace = EXE(scenario0);
// Green Lights
law38_sub1_1 = (trafficLightAhead.color == green) & (
    stoplineAhead(2) | junctionAhead(2)) & ~
    PriorityNPCAhead & ~PriorityPedsAhead;
law38_sub1_2 = F[0,2](speed > 0.5);
law38_sub1 = G (law38_sub1_1 -> law38_sub1_2);

// Yellow Lights
law38_sub2_1 = ((trafficLightAhead.color == yellow) & (
    stoplineAhead(0) | currentLane.number == 0)) -> (F
    [0,2] (speed > 0.5));
law38_sub2_2 = ((trafficLightAhead.color == yellow) &
    stoplineAhead(3.5) & ~stoplineAhead(0) & currentLane
    .number > 0) -> (F[0,3] (speed < 0.5));
law38_sub2 = G (law38_sub2_1 & law38_sub2_2);

// Red Lights
law38_sub3_1 = ((trafficLightAhead.color == red) & (
    stoplineAhead(2) | junctionAhead(2)) & ~(direction
    == right)) -> (F[0,3] (speed < 0.5));
law38_sub3_2 = ((trafficLightAhead.color == red) & (
    stoplineAhead(2) | junctionAhead(2)) & direction ==
    right & ~PriorityNPCAhead & ~PriorityPedsAhead) -> (
    F[0,2] (speed > 0.5));
law38_sub3 = G (law38_sub3_1 & law38_sub3_2);

law38 = law38_sub1 & law38_sub2 & law38_sub3;
trace |= law38;
```

Listing 1: Specifying Article #38 in LawBreaker

temporal expression (indicated by 'G' for '*always*'), which indicates that its two parts `law38_sub2_1` and `law38_sub2_2` should always be satisfied by the vehicle under test.

First, `law38_sub2_1` specifies that if the traffic light is yellow and the vehicle is on the stop line, then the vehicle should proceed through the junction. Article #38 does not state how quickly the car should move through the junction in this scenario: like most traffic laws, there is a degree of ambiguity that is expected to be resolved by common sense or practice. In LawBreaker, however, we need to be more precise so that our specification can serve as an oracle, so we interpret the law as requiring that the vehicle will '*eventually*' (F) move within the next d time steps. The variable d can be customised by the user; we set d to 2 here.

Second, `law38_sub2_2` specifies that if the traffic light is yellow and the vehicle is a safe distance away from the stop line, then the vehicle is expected to eventually ('F') stop within the next three time steps. The laws regarding green (`law38_sub1`) and red (`law38_sub3`) lights are defined in a similar manner using the temporal and Boolean operators of LawBreaker.

Ultimately, the goal of LawBreaker is to be able to automatically generate test scenarios in which these specifications are *violated*. Furthermore, it aims to find as many different ways of violating the laws as possible, e.g. by finding different counterexamples for each of the sub-expressions (e.g. `law38_sub2_1`). To illustrate this, consider the following two test cases generated by LawBreaker. In the first test case, the AV already reached the stop line when the traffic light turned from green to yellow, but hesitated for long enough that it actually crosses the intersection during a red light. Hence, the AV violates not only the `law38_sub2_1` but also `law38_sub3_1`. In the second test case, the AV rushed the yellow light and *caused an accident*. In this situation, the AV violated `law38_sub2_2` by rushing through even though there was enough distance to stop.

$$
\begin{aligned}
Formula &\longrightarrow BoolExpr \mid \text{'}\sim\text{'}\,Formula \mid Formula\ \text{'}\&\text{'}\ Formula \mid \\
&\quad Formula\ \text{'}|\text{'}\ Formula \mid Formula\ \text{'U'}\ Interval\ Formula \mid \\
&\quad \text{'G'}\ Interval\ Formula \mid \text{'F'}\ Interval\ Formula \mid \text{'N'}\ Formula \\
BoolExpr &\longrightarrow x_b \mid e_1 \oplus e_2
\end{aligned}
$$

Figure 3: Syntax of LawBreaker formulas, where x_b is a Bool variable; e_1 and e_2 are expressions; and $\oplus$ is a relational operator, i.e. $\oplus \in \{=, >, <, \leq, \geq\}$

3 SPECIFYING TRAFFIC LAWS

We introduce the syntax and semantics of our DSL for specifying driver-oriented traffic laws and present our main case study.

3.1 Syntax and Semantics

We first need the concept of a *trace* for understanding the syntax of our specification language. A trace is a sequence of scenes, and a scene is a snapshot of the world (i.e. the status of all vehicles, pedestrians, and so on). In LawBreaker, we offer a range of variables which allow us to extract relevant information from the scene, forming the building blocks of our specification language.

At the top-level, our specification language takes the form of temporal logic formulas. The syntax is shown in Figure 3, where U, G, F, and N respectively represent the temporal operators 'until', 'always', 'eventually', and 'next'. Furthermore, *Interval* is a real-time interval $[a, b]$ in which a and b are numerical expressions.

Intuitively, the formula f_1 U[a,b] f_2 (with f_1, f_2 derived from *Formula*) expresses that f_1 is true at every time point t of the trace until f_2 becomes true within the time interval $[t + a, t + b]$. Similarly, G[a,b]f_1 (resp. F[a,b]f_1) is true if the formula f_1 always (resp. eventually) holds in time interval $[t + a, t + b]$ for every time point t. Finally, N f_1 holds if f_1 is true in the next time point.

Temporal formulas are defined over Boolean expressions, which are built over several domain-specific variables related to the driver and its immediate surroundings. For example, the Bool variable `isTrafficJam` is true if there is a traffic jam ahead of the AV under test. Using this variable is more convenient, for example, than a 'global' approach which would require some quantification over all other NPC vehicles and a judgement based on the direction of travel and the position of the AV. As there are a range of variables in our language, we organise them into a few categories.

Car and Driving Status Variables. Car status variables can be used to describe properties involving the lights, engine, horn, and direction of the AV. The properties supported by LawBreaker are summarised in Table 1. These variables are self-explanatory (e.g. `hornOn` is true if and only if the horn is sounding) and are either of Bool or enumerated type.

Driving status variables can be used to describe the speed, acceleration, and braking status of the AV. Furthermore, there are Bool variables that capture manouevres that the AV is currently undertaking, e.g. changing lanes, overtaking another vehicle, or

Table 1: Car status variables in `LawBreaker`

Variable	Type	Remarks
highBeamOn	Bool	–
lowBeamOn	Bool	–
turnSignal	Enum	off, left, or right
fogLightOn	Bool	APIs not available in Apollo
hornOn	Bool	–
warningFlashOn	Bool	APIs not available in Apollo
gear	Enum	NEUTRAL, DRIVE, REVERSE, PARK, LOW, INVALID, or NONE
engineOn	Bool	–
direction	Enum	forward, left, right
toManual	Bool	True if and only if AV control passed to human operator

Table 2: Driving status variables in `LawBreaker`

Variable	Type	Remarks
speed	Number	Speed of ego vehicle (km/h)
acc	Number	Acceleration of ego veh. (m/s^2)
brake	Number	Braking percentage of ego veh. (%)
isChangingLane	Bool	–
isOverTaking	Bool	–
isTurningAround	Bool	–

Table 3: Road variables in `LawBreaker`

Variable	Type	Remarks
currentLane	Lane	–
speedLimit	SpeedLimit	–
streetLightOn	Bool	–
honkingAllowed	Bool	–
crosswalkAhead(n)	Bool	Within distance n
junctionAhead(n)	Bool	Within distance n
specialLocationAhead(n)	SpecialLocation	Within distance n
stoplineAhead(n)	Bool	Within distance n

turning around. Table 2 summarises them.

Road Variables. Road variables, summarised in Table 3, capture properties of the road the AV is currently driving on, e.g. whether or not honking is allowed, the street light is on, or whether a junction is within n units of distance ahead of the AV. Most of the variables are self-explanatory, but three of them are based on special types.

First, `currentLane` returns the lane that the AV is currently on. This is an object `lane` of type `Lane`, containing information such as the number of the current lane (`lane.number`), the side of the road that the lane is on (`lane.side`), and the allowed direction of travel (`lane.direction`). This can include values such as `left`, `right`, `UTurn`, and various other combinations (e.g. `forwardOrRight`).

Second, `speedLimit` returns an object `limit` of type `SpeedLimit`. This contains information such as the lower (`limit.lowerLimit`) and upper (`limit.upperLimit`) speed limits of the road. If a road does not have speed limits, the default values of these attributes will respectively be $-\infty$ and $+\infty$.

Finally, `specialLocationAhead(n)` returns an object `location` of type `SpecialLocation`. This contains attribute `location.type`, which can have one of the following values: `Railway_J`, `Bridge`, `SharpTurn`, `SteepSlope`, `Tunnel`, `ArchBridge`, `Slope`, `Flooded`, `OnewayRoad`, `Roundabout`, or `None` (i.e. none of the above).

Table 4: Signal variables in `LawBreaker`

Variable	Type	Remarks
stopSignAhead(n)	Bool	Within distance n
noUTurnSignAhead(n)	Bool	Within distance n
signalAhead	Enum	Common, Arrow, or None
trafficLightAhead	Signal	–

Table 5: Traffic variables in `LawBreaker`

Variable	Type	Remarks
PriorityNPCAhead(l/r)	Bool	Vehicle with right of way
PriorityPedsAhead(l/r)	Bool	Pedestrian with right of way
NPCAhead	NPC	–
NPCBack	NPC	–
NPCLeft	NPC	–
NPCRight	NPC	–
nearestNPC	NPC	–
NPCOpposite	NPC	–
isTrafficJam	Bool	–

Signal Variables. Signal variables, summarised in Table 4, allow for the specification of laws involving traffic lights and various signs (e.g. stop signs) at the junction an AV is approaching. Two of the variables are self-explanatory Bool types, but the other two consist of richer data. First, `signalAhead` is of enumerated type. The value `Common` indicates a traffic light in which all lights are circles, whereas the value `Arrow` indicates one in which (some of) the lights are arrows. `None` indicates that there is no traffic light ahead. Second, `trafficLightAhead` returns an object `signal` of type `Signal`. This consists of information such as the current colour of the traffic light (`signal.color`), which can be `yellow`, `green`, `red`, or `black`; whether or not the light is blinking (`signal.isBlinking`); and `signal.arrow`, which returns an object `arrow` of type `SignalArrow` that contains similar information to `Signal` objects but also the direction of the arrow.

Traffic Variables. Traffic variables, summarised in Table 5, are associated with other vehicles (NPCs) sharing the road with the AV, as well as any pedestrians crossing it. `PriorityNPCAhead(l/r)` indicates if there is an NPC vehicle ahead with priority over the ego vehicle, e.g. due to priority at junctions, or due to the NPC being an ambulance. The variable `PriorityPedsAhead(l/r)` indicates that there is a pedestrian with priority right of way ahead: pedestrians on a crosswalk, for example, have higher priority and are considered to be 'ahead' if they are within five metres of the ego vehicle. In both cases, the ego vehicle must not hinder the movement of the priority NPC/pedestrian when turning. Note that for the customisation of traffic laws across different countries, we can use `l/r` to represent the different sides of the driver position: `PriorityNPCAhead(l)` and `PriorityPedsAhead(l)` if the driver position of the country/region is at the left side, and `r` if it is at the right. By default, the driver position will be treated as left for both variables.

The variables `NPCAhead`, `NPCBack`, `NPCALeft`, `NPCRight`, `nearestNPC`, and `NPCOpposite` respectively represent the NPC vehicle in front of the ego vehicle, the one behind, the one on the left, the one on the right, the one that is closest, and the one that is opposite. These variables return objects `npc` of type of `NPC`, which contain information such as the speed of the NPC vehicle (`npc.speed`), the direction of the NPC vehicle (`npc.direction`), the type of the NPC

Table 6: Map variables in LawBreaker

Variable	Type	Remarks
weather	Weather	Current weather conditions
time	Time	Current day/time

vehicle (npc.type), as well as npc(n), which is true if the NPC is within n units of distance from the ego vehicle. Note that the value of npc.type can have one of the following values: bus, car, priorityVehicle, or None.

Map Variables. The map variables, shown in Table 6, are used to specify traffic laws related to environment conditions, e.g. the weather or time of day. The variable weather returns an object w of type Weather, consisting of information such as the degree of rain (w.rain, valued from 0 to 1), degree of fog (w.fog), or degree of snow (w.snow), and the current visibility in metres.

Semantics. Our specifications are interpreted over *execution traces* from the ADS. An execution trace π is a sequence of scenes, denoted as $\pi = \langle \theta_0, \theta_1, \ldots, \theta_n \rangle$. A scene θ is a tuple of the form $\theta = (f_0, f_1, \ldots, f_x)$ where f_i is the valuation of all of the above-mentioned variables.

Given a trace π, we write $\pi \vDash \Phi$ (resp. $\pi \nvDash \Phi$) to denote that Φ evaluates to be true (resp. false) given trace π. We use the standard definition of $\vDash$ for STL formulas (see e.g. [40]).

3.2 Case Study: Modelling China's Traffic Laws

Extending the illustrative example from Section 2, as our main case study, we examined all of the traffic laws in the *Regulations for the Implementation of the Road Traffic Safety Law of the People's Republic of China* [12]. We labelled each rule with the following flags: *relevant*, if the rule constrains an AV's behaviour in some way; *describable*, if it can be specified using LawBreaker; and *testable*, if the rule can potentially be tested in existing simulators (e.g. LGSVL). A summary of the labels for China's traffic laws is given in Table 7, and fully translated formulas for the describable laws can be found on our website [6]. (Our supplementary material [6] also includes detailed translations of Singapore's traffic laws, demonstrating the generality of the language.)

While all laws are relevant, some of them are not describable in LawBreaker. A typical example is Article #65, which requires drivers to "obey the instructions" of ferry management personnel (this is too vague for our language to describe). A number of rules are not testable due to lack of support in the underlying simulator. For example, Article #43 regulates the behaviour of vehicles when crossing a railway track. While our language and fuzzer can be extended to cover such situations, railway tracks are not yet supported in the current maps of the simulator. Additional details are presented later, in Section 5.2, as part of our evaluation.

We present two of the translated traffic laws to highlight how the language is used.

Article #42: Yellow Lights. Article #42 stipulates that when a vehicle passes over a junction with flashing yellow light, it needs to ensure safety when passing through:

Table 7: Summary of Chinese traffic rules

Category	Metric	Relevant	Describable	Testable
General	Count	8	8	**3**
	Percent	100%	100%	**37.5%**
Vehicles	Count	40	37	**21**
	Percent	100%	92.5%	**52.5%**
Highway	Count	12	12	**0**
	Percent	100%	100%	**0%**
Others	Count	0	0	0
	Percent	-	-	-

"The flashing warning signal light is a yellow light that continues to flash, reminding vehicles and pedestrians to pay attention when passing through, and pass after confirming safety."

This is an example of an ambiguous (or vague) traffic law that requires a specific formalisation in LawBreaker. In particular, we can specify it as follows:

```
proposition1 = trafficLightAhead.color==yellow &
    trafficLightAhead.blink &  (stoplineAhead(10) |
    junctionAhead(10));
proposition2 = F[0,2]((speed<20) U ~NearestNPC(15));
law42 = G(proposition1 -> proposition2);
```

The above translation expresses that when a flashing warning signal is v_1 meters ahead, the ego vehicle should move at a speed of less than v_2 km/h until there are no other vehicles within v_3 meters in the coming v_4 time steps. The value of variables v_1, v_2, v_3, v_4 can be customised by the user: here, we instantiated them with the values 10, 20, 15, and 2.

Note that the user can also customise these traffic laws in their own way, for instance, users can add constraints regarding pedestrians ahead by using the signal variable PriorityPedsAhead in the formula.

Article #52: Priority. Article #52 stipulates priority issues regarding vehicles, in particular, that the vehicle should give way to the vehicles with higher priority:

"When a motor vehicle passes through an intersection that is not controlled by traffic lights or commanded by traffic police, in addition to complying with the provisions of Article #51 (2) and (3), it shall also comply with the following provisions:

(1) If there are traffic signs and markings, let the party with priority go first;
(2) If there is no traffic sign or marking control, stop and look at the intersection before entering the intersection and let the traffic on the right road go first;
(3) Turning motor vehicles let vehicles going straight go first;
(4) A right-turning motor vehicle driving in the opposite direction should let the left-turning vehicle go first."

This is an example of how our driver-oriented specification approach can lead to a simpler specification than an equivalent globally specified property (as in AVUnit [3]). In LawBreaker, we can describe sub-rules 2–4 of Article #52 in the following way:

```
prop1 = signalAhead==None & junctionAhead(0.5);
prop2 = F[0,2]((speed<0.5)U(~PriorityNPCAhead));
law52 = G(prop1 -> prop2);
```

The above translation expresses that when there is an intersection without traffic lights ahead, the ego vehicle is expected to stop

until there is no vehicle with higher priority ahead. In order to simplify the specification of this traffic law, we utilise the signal variable `PriorityNPCAhead` to check whether there is a priority vehicle ahead. The design principle behind `PriorityNPCAhead` can be found in Section 3.1: ultimately, this signal variable allows a direct translation of the traffic law.

4 THE LAWBREAKER FUZZING ENGINE

In the real world, traffic laws can be violated in multiple different ways. For instance, as shown in Section 2, there are two ways to violate Article #38's yellow light sub-rule: failing to move when the ego vehicle has already reached the stop line, and failing to stop when the ego vehicle is at a safe stopping distance before it. Both of these violations are interesting, but for very different reasons: the first leads to an AV that is driving less efficiently than it (legally) might, whereas the second is quite dangerous and could increase the likelihood of a traffic accident. In general, we need a fuzzing approach that can find as many different ways of violating a specification as possible, as multiple test cases will help to pinpoint the key problems in the ADS.

In this section, we present a fuzzing algorithm based on the idea of: (1) identifying the different possible ways of violating a law specification Φ in our language, then (2) searching for concrete test cases that come 'closer' (as measured by a quantitative semantics) to violating Φ in the different ways that were identified.

4.1 Specification Violation Coverage

Given a law specification Φ, we write $\Theta(\Phi)$ to denote a set of constraints that represents different ways in which Φ might be violated. Formally, $\Theta(\Phi)$ is a set of formulas which satisfies the following proposition:

Proposition 4.1. *Let Φ be an STL formula and π a trace. Then:*

$$\forall \xi \in \Theta(\Phi).\ \pi \vDash \xi \implies \pi \nvDash \Phi$$

□

In `LawBreaker`, $\Theta(\Phi)$ is computed as follows:

$$\begin{aligned}
\Theta(\mu) &= \{\neg\mu\}, \textit{where } \mu \textit{ is a Boolean expression}\\
\Theta(\theta_1 \wedge \theta_2) &= \Theta(\theta_1) \cup \Theta(\theta_1)\\
\Theta(\theta_1 \vee \theta_2) &= \{x \wedge y \mid x \in \Theta(\theta_1) \wedge y \in \Theta(\theta_2)\}\\
\Theta(\neg\theta_1) &= N(\theta_1)\\
\Theta(\Box_I \theta_1) &= \{\Diamond_I\ \theta \mid \theta \in \Theta(\theta_1)\}\\
\Theta(\Diamond_I \theta_1) &= \{\Box_I\ \theta \mid \theta \in \Theta(\theta_1)\}\\
\Theta(\theta_1 \mathcal{U}_I \theta_2) &= \{x\ \mathcal{U}_I\ y \mid x \in \Theta(\neg\theta_1 \vee \theta_2) \wedge y \in \Theta(\theta_1 \vee \theta_2)\}\\
&\quad \cup \{x \wedge y \mid x \in \Theta(\theta_1) \wedge y \in \Theta(\theta_2)\}\\
\Theta(\bigcirc\theta_1) &= \{\bigcirc x \mid x \in \Theta(\theta_1)\}
\end{aligned}$$

where $N(\Phi)$ represents different ways in which Φ might be satisfied, i.e., $N(\Phi)$ is a set of formulas satisfying the following condition:

$$\forall \xi \in N(\Phi).\ \pi \vDash \xi \implies \pi \vDash \Phi$$

It is systematically computed as follows:

$$\begin{aligned}
N(\mu) &= \{\mu\}, \textit{where } \mu \textit{ is a Boolean expression}\\
N(\theta_1 \wedge \theta_2) &= \{x \wedge y \mid x \in N(\theta_1) \wedge y \in N(\theta_2)\}\\
N(\theta_1 \vee \theta_2) &= N(\theta_1) \cup N(\theta_2)\\
N(\neg\theta_1) &= \Theta(\theta_1)\\
N(\Box_I \theta_1) &= \{\Box_I\ \theta \mid \theta \in N(\theta_1)\}\\
N(\Diamond_I \theta_1) &= \{\Diamond_I\ \theta \mid \theta \in N(\theta_1)\}\\
N(\theta_1\ \mathcal{U}_I\ \theta_2) &= \{x\ \mathcal{U}_I\ y \mid x \in N(\theta_1) \wedge y \in N(\theta_2)\}\\
N(\bigcirc\theta_1) &= \{\bigcirc\theta \mid \theta \in N(\theta_1)\}
\end{aligned}$$

Proposition 4.1 can be proven by structural induction. The detailed proof can be found in our supplementary materials [6].

Example 4.2. *The value of $\Theta(\Phi)$ is calculated recursively. For example, given a specification $\Phi = \Box((a \vee b) \rightarrow c)$, before the calculation of Θ, we first pre-process Φ to get $\Phi' = \Box(\neg(a \vee b) \vee c)$. The formula Φ is equivalent to Φ'. Then, we can get $\Theta(\Phi)$ as follows:*

(1) Calculation of the primitive elements:
$\Theta(c) = \{\neg c\}, N(a) = \{a\}, N(b) = \{b\}$
(2) Given $N(a) = \{a\}$ and $N(b) = \{b\}$: $N(a \vee b) = \{a, b\}$
(3) Given $N(a \vee b)$: $\Theta(\neg(a \vee b)) = \{a, b\}$
(4) Given $\Theta(\neg(a \vee b))$ and $\Theta(c)$:
$\Theta(\neg(a \vee b) \vee c) = \{a \wedge \neg c, b \wedge \neg c\}$
(5) Given $\Theta(\neg(a \vee b) \vee c)$:
$\Theta(\Box(\neg(a \vee b) \vee c)) = \{\Diamond(a \wedge \neg c), \Diamond(b \wedge \neg c)\}$
(6) The final result is $\Theta(\Phi) = \{\Diamond(a \wedge \neg c), \Diamond(b \wedge \neg c)\}$.

i.e. we can 'cover' the different ways of violating the original specification Φ by finding traces that satisfy $\Diamond(a \wedge \neg c)$ and $\Diamond(b \wedge \neg c)$.

4.2 Quantitative Semantics

Given a formula Φ, the overall idea of our fuzzing algorithm is to systematically generate test cases to violate each STL formula $\varphi \in \Theta(\Phi)$, if feasible. We thus first adopt a quantitative semantics for our specification language, which allows us to iteratively generate test cases that come 'closer' to violating a given formula.

The quantitative semantics is adopted from [14, 40, 46], which, intuitively speaking, defines the semantics of a formula φ with respect to a trace π in the form of a *robustness value*, i.e. a number representing how far φ is from being satisfied by π. Our algorithm then attempts to maximise this number so as to generate a violation.

Definition 4.3 (Quantitative Semantics). Given a trace π and a formula φ, the quantitative semantics is defined as the robustness degree $\rho(\varphi, \pi, t)$ where t is the time step:

$$\begin{aligned}
\rho(\mu, \pi, t) &= f(\pi)(t),\\
\rho(\neg\varphi, \pi, t) &= -\rho(\varphi, \pi, t),\\
\rho(\varphi_1 \wedge \varphi_2, \pi, t) &= \min\{\rho(\varphi_1, \pi, t), \rho(\varphi_2, \pi, t)\},\\
\rho(\varphi_1 \vee \varphi_2, \pi, t) &= \max\{\rho(\varphi_1, \pi, t), \rho(\varphi_2, \pi, t)\},\\
\rho(\varphi_1\ \mathcal{U}_I\ \varphi_2, \pi, t) &= \sup_{t' \in t+I} \min\{\rho(\varphi_2, \pi, t'), \inf_{t'' \in [t, t']} \rho(\varphi_1, \pi, t'')\},\\
\rho(\Diamond_I \varphi, \pi, t) &= \sup_{t' \in t+I} \rho(\varphi, \pi, t'),\\
\rho(\Box_I \varphi, \pi, t) &= \inf_{t' \in t+I} \rho(\varphi, \pi, t'),\\
\rho(\bigcirc\varphi, \pi, t) &= \rho(\varphi, \pi, t+1).
\end{aligned}$$

If $\rho(\varphi, \pi, t)$ is equal to or greater than 0, we successfully found a way to violate Φ. Note that $\rho(\varphi, \pi)$ is treated as $\rho(\varphi, \pi, 0)$.

Example 4.4. *Given a traffic law* $\Phi = \Box(speed < 80)$ *which means the speed of the ego vehicle should always be less than* $80km/h$*, suppose the speed over a trace* π *is* $\{(t, s) : (t_0, 0), (t_1, 0), (t_2, 0.5), (t_3, 1.8), (t_4, 4.5), (t_5, 7.9), (t_6, 10.9), \ldots, (t_{39}, 85), \ldots\}$ *where the maximum value of speed is* $85km/h$ *at time step* t_{39}*, and the relevant specification* $\varphi \in \Theta(\Phi)$ *is* $\varphi = \Diamond(speed > 80)$*, then we have* $\rho(\varphi, \pi) = \rho(\varphi, \pi, 0) = max_{t' \in [0, |\pi|]}(speed(t') - 80) = 5 > 0$*. It means the specification* φ *is satisfied by* π *which leads to a violation of the traffic law* Φ.

4.3 Genetic Encoding for Scenarios

Our fuzzing algorithm is based on a genetic algorithm (GA) [42], and requires an appropriate genetic encoding for the targeted scenario description language, as well as a customisation of the crossover and mutation operators.

In this work, we adopt the AVUnit [3] scenario description language, which describes a test case in terms of the ego vehicle, NPC vehicles, pedestrians, static obstacles, and the environment (e.g. weather). We encode scenarios in terms of their operable parameters. For the ego vehicle, the operable parameter is its starting point. For NPCs, all parameters are operable except the speed at their destinations (which is always zero). Finally, for static obstacles and the environment, all parameters are operable. Note that in valid encodings, vehicles must always be positioned within a lane or junction area, pedestrians may move around empty regions of the map, and obstacles can be placed anywhere (e.g. a basketball on the road, or even a meteorite on the crosswalk).

For parameters with continuous values (e.g. position, speed, and weather), we apply Gaussian mutation. We also apply the clipping function to avoid invalid values.

For each pair of test cases, the genetic crossover operation can only be done in the same category (e.g. position, NPC type). In theory, we can perform crossover for each category, but to avoid generating infeasible scenarios, we do not perform crossover on the chromosomes encoding vehicle and pedestrian positions.

Note that since the crossover and mutation operations are always limited to the valid space of the operable parameters, the newly generated test cases are always valid.

4.4 Fuzzing Algorithm

We are now ready to present the overall fuzzing algorithm of `LawBreaker`, which directly utilises $\Theta(\Phi)$ (i.e. the different possible ways to violate Φ) and $\rho(\xi, \pi)$ (i.e. how 'close' we are to violating the formula ξ). Note that $\xi \in \Theta(\Phi)$.

Our fuzzing approach is detailed in Algorithm 1. First, we generate some initial test cases randomly then initialise $Seeds_r$ as empty and $Robust_r$ as $-\infty$. Note that $Seeds_r$ and $Robust_r$ are mappings from the remaining formulas in Θ_r to the corresponding test cases and robustness scores. For every test case in a generation, we execute it to obtain trace π and compute the robustness $\rho(\xi, \pi)$ for all the uncovered $\xi \in \Theta_r$. We remove ξ once it is satisfied, i.e. $\rho(\xi, \pi_i) \geq 0$, and add the corresponding scenario s_i to the output set Γ. If some ξ is not satisfied, we update $Seeds_r(\xi)$ and $Robust_r(\xi)$ when the current trace is closer to the satisfaction of ξ, i.e. $\rho(\xi, \pi_i) > Robust_r(\xi)$.

Algorithm 1: `LawBreaker` Fuzzing Algorithm

Input: Φ, n (population size), and M (maximal generations)
Output: A test suite Γ

Let $\Theta_r = \Theta(\Phi)$ be the set of uncovered formulas in $\Theta(\Phi)$;
Let $Seeds_r$ be a mapping from Θ_r to tests, initially empty;
Let $Robust_r$ be a mapping from Θ_r to robustness, initially $-\infty$;
Let $G = \{s_1, \cdots, s_n\}$ be a set of randomly generated tests;
Let Γ be an empty set;
while Θ_r *is not empty and not timeout* **do**
  for *each* s_i *in* G **do**
    Execute s_i via simulation and obtain trace π_i;
    for *each* ξ *in* Θ_r **do**
      Compute the robustness $\rho(\xi, \pi_i)$;
      if $\rho(\xi, \pi_i) \geq 0$ **then**
        Remove ξ from Θ_r; Add s_i into Γ;
      end
      else if $\rho(\xi, \pi_i) > Robust_r(\xi)$ **then**
        Set $Seeds_r(\xi)$ to be s_i;
        Set $Robust_r(\xi)$ to be $\rho(\xi, \pi_i, t)$;
      end
    end
  end
  Set G to be a set of new test cases generated based on $Seeds_r$ through selection, mutation and crossover;
end
return Γ

After executing and processing all of the test cases of a population, we generate the population for the next generation based on $Seeds_r$ and $Robust_r$. Since the size of $Seeds_r$ is equal to the size of Θ_r, it may be larger or smaller than the population size, and thus we select parents as follows. We first sort $Seeds_r$ according to $Robust_r$ in descending order. Note that the greater the robustness value is, the more likely the test case leads to a new violation of the traffic law Φ. To add some uncertainty, we first choose an individual from the first half of the population, and then we select an individual from the overall population by random sampling. From these two selected individuals, the individual with the higher score of robustness is chosen. Note that the higher the score, the closer the test case is to the targeted violation. We repeat this selection process until a given number of the parent population is selected. (This number can be chosen by the user.)

With the parents set selected, we apply crossover and mutation as defined in Section 4.3 to obtain the next generation. This process is repeated until all the elements in set $\Theta(\Phi)$ are covered or the maximal number of generations has been reached. Note that the maximal number of generations M is defined by the user.

5 IMPLEMENTATION AND EVALUATION

In this section, we present our implementation and evaluation of `LawBreaker` based on an existing popular simulation framework.

5.1 Implementation

Implementing `LawBreaker` for a given ADS and simulator requires the completion of the following three steps: (1) construction of a

bridge for collecting messages from the ADS and spawning scenarios in the simulator; (2) implementation of a *library* for converting those messages into signals for the traffic law language in Section 3; and (3) implementation of the *fuzzing engine* in Section 4. Our source code of all three components for Apollo+LGSVL is available online [6].

In this work, we implemented a bridge for the Apollo 5.0 and Apollo 6.0 ADSs. Our bridge retrieves messages from the ADS that include the driving status of the ego vehicle (e.g. speed, position, high beam) and the environment (e.g. status of NPC vehicles and nearby traffic signals) at different time steps. Currently, the perception part of both Apollo versions is still under development, so we follow the recommendation of the vendor and use the ground truth as the input of the `PerceptionObstacle` module. Our bridge allows for all the operable parameters mentioned in Section 4.3 (e.g. trajectories of NPC vehciles and pedestrians) to be translated to API calls in the simulator, thus spawning scenarios based on our genetic encoding.

With the bridge retrieving the original messages from the ADS, we then use our library to translate these messages into the signal variables described in Section 3. Some signal variables are quite intuitive. For instance, the signal variables `speed`, `acc`, `brake` are the speed, acceleration, and brake percentages of the ego vehicle, and we obtain these through a simple analysis of ADS messages from different time steps. However, some signal variables require more complex processing. For example, it takes a few steps to calculate the value of signal `PriorityPedsAhead` at time step `t`. First, we calculate the 'area ahead' with respect to the current position and direction of the ego vehicle. Then, we check whether there is a pedestrian within that area based on the positions of pedestrians. If so, we check whether the pedestrian is likely to cross based on their distance to a crosswalk. If both are satisfied, then the value of the signal `PriorityPedsAhead` at that time step is set to be true, otherwise false.

Finally, we implemented the fuzzing engine as described in Section 4.4. Our implementation supports six kinds of mutations, i.e., mutation of position, speed, time, weather, NPC vehicle type, and pedestrian type. Furthermore, we embedded the tool *RTAMT* [46] to compute the robustness of the specifications with respect to the trace obtained from the bridge.

5.2 Evaluation

In the following, we conduct multiple experiments to answer our key Research Questions (RQs). Since `LawBreaker` is designed for the description and evaluation of traffic laws across different countries, we evaluate it from three aspects: versatility, effectiveness, and efficiency, which correspond to our three RQs.

RQ1: Can we test AVs against traffic laws using LawBreaker? To answer this question, we systematically examined all Chinese traffic laws related to AVs and determined whether or not they were describable and testable (as per Section 3.2) using `LawBreaker`. In addition, we examine whether the same laws can be expressed using alternative specification approaches provided by other frameworks.

A number of existing works [3, 19, 39, 50, 51] propose methods of evaluating AVs under different oracles, and we compare against the ones that are capable of specifying (at least some) traffic laws. In particular, we compare against formalisations in Isabelle/HOL (FIH) [51], for Machine Interpretability (FMI) [19], in Temporal Logic (FTL) [39], for Accountability (FA) [50], and finally, AVUnit's own test engine based on global specifications [3].

Table 8: Testing AVs, where ×, Δ, and √ means no support, limited support and full support respectively

Traffic Laws		FIH	FMI	FTL	FA	AVUnit		LawBreaker		
		D	D	D	D	D	T	D	T	Why $\neg D \vee \neg T$
Law38 sub1-3		×	×	×	×	Δ	Δ	√	√	-
Law40-43		×	×	×	×	×	×	√	×	Lack Map Support
Law44		×	Δ	Δ	×	×	×	√	√	-
Law45	sub1	×	√	√	Δ	√	√	√	√	-
	sub2	×	√	√	Δ	√	√	√	√	-
Law46	sub1	×	×	Δ	×	×	×	√	×	Lack Map Support
	sub2	×	×	×	×	×	×	√	Δ	Lack Map Support
	sub3	×	×	×	×	×	×	√	√	-
	sub4	×	×	×	×	×	×	√	×	Lack Map Support
Law47		√	Δ	Δ	×	×	×	√	√	-
Law48 sub1-2		×	Δ	Δ	Δ	×	×	√	×	Lack Map Support
Law48 sub3-4		×	×	×	×	×	×	×	×	Vague
Law48	sub5	×	×	×	×	×	×	√	×	Lack Map Support
Law49		×	×	×	×	×	×	√	×	Lack Map Support
Law50		×	×	Δ	×	×	×	√	√	-
Law51 sub1-2		×	×	×	×	×	×	√	×	Lack Map Support
Law51	sub3	×	×	×	×	×	×	√	√	-
Law51 sub4-7		×	×	×	×	Δ	Δ	√	√	-
Law52	sub1	×	×	×	×	×	×	√	×	Lack Map Support
Law52 sub2-4		×	Δ	×	×	Δ	Δ	√	√	-
Law53		×	×	√	×	Δ	Δ	√	√	-
Law57 sub1-2		×	×	×	×	×	×	√	√	-
Law58		×	×	×	×	×	×	√	√	-
Law59		×	×	×	×	×	×	√	√	-
Law62	sub1	×	×	×	×	×	×	√	×	Lack Sensors
Law62	sub4	×	×	×	×	×	×	√	×	Lack Map Support
Law62	sub8	×	×	×	×	Δ	×	√	×	Lack Map Support
Law63 sub1-3		×	×	×	×	×	×	√	×	Lack Map Support
Law64		×	×	×	×	×	×	√	×	Lack Map Support
Law65		×	×	×	×	×	×	×	×	Vague
Law78		×	Δ	Δ	Δ	√	×	√	×	Lack Map Support
Law 79-82		×	Δ	Δ	×	×	×	√	×	Lack Map Support
Law84		×	×	×	×	Δ	×	√	×	Lack Map Support

Table 8 presents the results of our evaluation. Here, **D** means whether the framework allows description of the traffic law, and **T** means whether the framework can test the specific traffic law with the support of existing simulators. We observe that `LawBreaker` supports most of the relevant traffic laws and outperforms the existing works in this aspect, largely due to our driver-oriented language that allow specifications to be scenario-independent. For example, for traffic lights, AVUnit's own test engine (based on global specifications) requires the user to be familiar with the map and to formulate specifications based on the IDs of every traffic light, the positions of every vehicle, and so on. Even more problematic is that users have to write a different specification for every specific scenario since the map is different. `LawBreaker` solves this problem by making use of the driver-oriented signal variable `trafficLightAhead`, which makes it independent from the scenarios.

Focusing on `LawBreaker`, the last column of Table 8 summarises the reason why `LawBreaker` is unable to describe or test a given Chinese traffic law. There are several reasons. First, it may be because the law is irrelevant to AVs (e.g. they regulate the behaviour of pedestrians rather than the AVs); we do not list these in the table. Second, it may be because the law is hard to evaluate or quantify. For instance, sub4 of Law48 [12], which regulates priority on mountain roads, is rather vague. Third, the law cannot be tested due to

the limitation of existing maps. For example, Law40 and Law41 regulate the behaviour of vehicles when facing traffic lights with arrow lights cannot be tested, since LGSVL does not support traffic lights with arrow lights (this is a limitation of the simulator, rather than `LawBreaker`). Finally, some laws cannot be tested due to the lack of certain sensors. For example, sub1 of Law62 requires that the doors and compartments must be closed when driving. However, there is currently no sensor in the ADS for detecting the status of doors and compartments.

To summarise, `LawBreaker` is able to specify most of the relevant laws. The main reason why some laws cannot be tested is the limitation of the underlying simulators, i.e. those laws can be tested in the future once sufficient map and sensor support are provided.

RQ2: How effective is LawBreaker at generating violations of laws? To answer this question, we systematically apply our fuzzing algorithm to test all the testable laws. The results are summarised in Table 9. Note that there are two different versions of an ADS driver being tested: *Apollo5.0* and *Apollo6.0*, which are the two latest versions of ADSs developed on the Apollo platform. The *Violations* and *Accidents* in the table denote whether the driver violates the traffic law and whether there are accidents due to the violations of the law. Note that we mark √ for a traffic law Φ if and only if accidents happened in the trace π and $\pi \not\models \Phi$. Since ours is the first work which is capable of generating law-breaking test cases, we have no baseline to compare with in this experiment.

As can be seen from the table, `LawBreaker` is able to trigger violations of most of the laws (sometimes in multiple ways). In summary, `LawBreaker` is able to find violations of 14 different Chinese traffic laws by Baidu Apollo. Among the test cases generated by our framework, 173 of them not only violate the laws but also *cause accidents*. Furthermore, *Apollo6.0* violates more traffic laws and results in more accidents than *Apollo5.0*. While this is surprising, a close investigation shows that *Apollo6.0* drives more aggressively than *Apollo5.0* since the *Apollo6.0* uses a deep learning model, while *Apollo5.0* is completely controlled through a program.

In the following, we categorise the identified issues and present examples in each category. Video recordings of all the identified issues are available at [6]. Note that we reran the corresponding test cases at least 3 times to ensure that all issues are reproducible.

Dangerous behaviours. The AV may break a law and result in dangerous behaviour. For instance, it might rush at a yellow light (violating Article #38) and cause accidents. On the other hand, it might also hesitate at a yellow light and cross the junction at a red light, i.e. although the AV reaches the stop line when the traffic light turned from green to yellow and is expected to go across, it hesitates at the yellow light and crosses the intersection eventually at the red light. In another instance, the AV may fail to complete overtaking a large vehicle and cause accidents. In this situation, the AV is trying to overtake a large turning vehicle (e.g. bus). But the ego vehicle wrongly estimates the distance to the large vehicle and accelerates, which causes collisions. The ego vehicle violates both Article #44 and Article #47 in this situation. Moreover, while the AV is expected to drive slowly with caution in heavy rain or fog, it ignores the weather condition and drives at a high speed (violating sub-rule3 of Article #46). We remark that some of these behaviours do not result in accidents and thus would be missed by existing approaches [3, 37]. They are nonetheless behaviours that should be investigated and corrected.

Table 9: Violations of Chinese traffic laws

Traffic Laws		Violations		Accidents		Content
		5.0	6.0	5.0	6.0	
Law38	sub1	√	√	×	√	green light
	sub2	√	√	√	√	yellow light
	sub3	√	√	×	√	red light
Law44		√	√	√	√	lane change
Law45	sub1	×	×	×	×	speed limit
	sub2	×	×	×	×	speed limit
Law46	sub2	×	√	×	×	speed limit
	sub3	√	√	√	√	speed limit
Law47		√	√	√	√	overtake
Law50		×	×	×	×	reverse
Law51	sub3	×	√	×	×	traffic light
	sub4	√	√	√	√	traffic light
	sub5	√	√	×	√	traffic light
	sub6	×	×	×	×	traffic light
	sub7	×	×	×	×	traffic light
Law52 sub2-4		×	×	×	×	priority
Law53		×	×	×	×	traffic jam
Law57	sub1	√	√	×	×	left turn signal
	sub2	√	√	×	×	right turn signal
Law58		√	√	√	√	warning signal
Law59		√	√	√	√	signals
Law62	sub8	×	×	×	×	honk

Inefficiency. The AV may break a law and result in inefficient driving. For instance, it might remain stationary while the traffic light is green. It might also hesitate at a yellow light—although it is safe to cross—until the traffic light turns red. Furthermore, it might fail to overtake a stationary vehicle ahead at an intersection. That is, there is a static NPC vehicle ahead and the traffic light turns to green. The AV is expected to overtake the static NPC vehicle to continue the journey. However, it remains stationary and fails to overtake (violating Article #38). Lastly, it may fail to make a necessary lane change and never reach the destination. In this situation, we set a destination that can be reached by a lane change after crossing the intersection ahead. However, the AV plans an unusual route, and keeps looping around and never reaches the destination.

RQ3: How efficient is LawBreaker at generating test cases? Since there is no existing framework to support the evaluation of traffic laws, we compare our fuzzing algorithm against a fuzzing algorithm based on random generation. For our fuzzing algorithm, we set the initial population to 20 and the number of generations to 20, resulting in 420 test cases in one run. For random generation, we randomly generate 420 tests cases for each run. We run our fuzzing algorithm and random generation four times to reduce the effect of randomness, and the results are summarised in Table 10. We evaluate *Apollo6.0* and *Apollo5.0* under all the testable traffic laws shown in Table 9. According to the definition of $\Theta(\Phi)$ in Section 4.1, there are 82 possible violations. We compare our fuzzing algorithm with random generation in three different scenarios. Overall, we provide ten AVUnit scenario scripts at [6], and use three of the scenarios for fuzzing because they are common real-world scenarios that happen to be associated with complex traffic laws (e.g. vehicle behaviour at junctions, overtaking).

Table 10: Violation coverage of different drivers

Scenario	Driver	Alg.	R1	R2	R3	R4	Avg
S1	Apollo6.0	Ours	27/82	25/82	21/82	27/82	**25**
		Rand	26/82	23/82	15/82	21/82	**21.25**
S2	Apollo6.0	Ours	23/82	24/82	26/82	27/82	**25**
		Rand	22/82	22/82	15/82	22/82	**20.25**
S3	Apollo6.0	Ours	24/82	22/82	25/82	23/82	**23.5**
		Rand	15/82	15/82	23/82	22/82	**18.75**
S1	Apollo5.0	Ours	27/82	22/82	22/82	23/82	**23.5**
		Rand	22/82	21/82	21/82	21/82	**21.25**
S2	Apollo5.0	Ours	17/82	16/82	17/82	15/82	**16.25**
		Rand	15/82	15/82	14/82	15/82	**14.75**
S3	Apollo5.0	Ours	25/82	24/82	24/82	25/82	**24.5**
		Rand	25/82	23/82	24/82	23/82	**23.75**

- *S1*: In this scenario, there is a T-junction with traffic lights ahead. There are four NPC vehicles in the scenario. The ego vehicle is expected to cross the intersection safely.
- *S2*: In this scenario, there are a few static and low-speed NPC vehicles ahead, and the ego vehicle is expected to overtake these static vehicles to reach the destination. There are five NPC vehicles in the scenario.
- *S3*: In this scenario, there is an intersection with traffic lights ahead and the lanes are of two opposite directions. There are five NPC vehicles in the scenario.

As can be seen from Table 10, our fuzzing algorithm outperforms random generation with respect to both versions of Apollo, showing its utility for automatic testing. Furthermore, when comparing driving strategies, *Apollo6.0* is more inclined to aggressive ones than *Apollo5.0*, leading to more violations of traffic laws for *Apollo6.0*. For the four runs of the above three scenarios, we generate 77 scenarios that can cause accidents for *Apollo5.0* and 96 for *Apollo6.0*. As mentioned before, the implementation of deep learning for the decision process of *Apollo6.0* is the main reason for this difference.

Threats to Validity. Due to the nature of simulation-based testing, there are threats to the validity of the discovered issues. For instance, some issues may only occur because of the latency between the simulator and the ADS.

To solve this problem, Apollo itself has some built-in mechanisms to handle these situations such as the "estop" command to stop the vehicle. Moreover, we have the following strategies to reduce the false-positive rate. First, all the found issues are repeated at least three times to ensure reproducibility. Second, the information we use for specifications is exactly the same as the ADS gets. In this way, even when there is a delay which causes the problem, we do not blame the ADS for it. Third, we make sure the devices for simulations are in good condition (e.g. well-connected, sufficient memory). Despite these measures, in general, we cannot rule out that a discovered problem may be due to the simulator. Nonetheless, uncovering such a problem may still be helpful for improving the system as a whole.

6 RELATED WORK

Critical Scenario Generation. A scenario for AV testing consists of static parameters (e.g. time, weather) and dynamic parameters (e.g. trajectory of vehicles and pedestrians). The main goal of existing works about AV testing is to generate scenarios that can expose vulnerabilities of AVs. We divide existing works into two groups: *recreating real-world scenarios*, and *generating new scenarios*.

The first group of works explores how to recreate real-world scenarios. TNO [48] provides a dataset containing 6000 kilometers of driving on public roads and promotes the development of scenario-mining algorithms. AC3R [26], and DEEPCRASHTEST [9] reconstruct car crashes to evaluate ADSs based on the scenario data from the police reports and accident videos respectively. K-medoids [47] focuses on recreating scenarios at T-and four-legged junctions based on the recordings of junction crashes in the UK. Recreating unsafe cut-ins based on human driver lane change behaviour is another way of accelerating the evaluation of AVs [57]. Extracting features of real-world scenarios to evaluate ADSs by comparing them with human drivers is also a solution [52].

The second group of works explores how to generate critical scenarios and defines the criticality of the scenario differently. AVFuzzer [37] proposes an autonomous way to generate critical scenarios by fuzzing. The fuzzing algorithm of AVFuzzer is optimised with respect to (only) the distance from other NPC vehicles, i.e looking for scenarios that are likely to cause collisions. NADE [24] automatically generates scenarios that are natural and critical at the same time based on the data collected from a real-world dataset. The criterion for evaluating whether a scenario is critical or not in NADE is the distance from other vehicles. Similarly, 'no collision' is also the criteria of Rule-based Searching [41], Evolutionary-Algorithm-based Generation [31], and CMTS [15]. A few works explore critical criteria beyond 'no collision'. Hungar [30], for example, defines the criticality of scenarios using a calculation over several harmful events-related variables. The Baidu group proposes a coverage-based feedback mechanism [29] that takes the coverage of the driving area of the map as the criterion, i.e. covering a larger driving area indicates a better scenario. Hauer and Schmidt [28] explore how to automatically or manually generate test cases that cover all categories and model this problem as a Coupon Collector's problem. PlanFuzz [56] focuses on overly-conservative ADS behaviours and checks whether the ego vehicle stops in safe conditions. Mullins defines a 'near-miss' by whether or not the scenario causes the AV to be at the boundary between distinct performance modes [43, 44]. Althoff and Lutz [8] define the criticality of the scenario by the size of the passable area, and generate critical scenarios by minimising the area. Beglerovic et al. [10] induce a cost function from the specification of scenarios.

Although existing works propose different methods to generate scenarios for the evaluation of AVs, they focus on weak oracles and they lack a set of systematic time-tested oracles for AV testing. In this work, we evaluated AVs under traffic laws and propose the `LawBreaker` fuzzing algorithm to generate 'critical' scenarios that are likely to violate the traffic laws in different ways.

Formalisation of Specifications. Existing works on robotic motion plan have implemented STL for describing complex oracles, e.g. [20–22, 27, 32–36, 38, 54]. These works demonstrate the relevance of STL for motion-related specifications. Existing STL-based AV-related specification languages, e.g. [3, 18, 55], cannot describe traffic laws. For example, VERIFAI [18] and MLSTL [55] focus on the evaluation of particular modules (e.g. perception) and fail to support the evaluation of the overall behaviour of ADSs. Although

AVUnit [3] proposes a concrete way to describe scenarios and a general specification language based on STL for evaluation of ADSs, AVUnit [3] describes all the specifications in a global perspective which is not suitable for expressing traffic laws via driver-oriented specifications. Some existing works [11, 13, 19, 39, 50, 51] provide some formalisation methods for traffic laws. Rulebook [11, 13] describes traffic laws by connecting atomic rules. The four formalisation methods [19, 39, 50, 51] propose different ways to describe traffic laws, but are limited to specific traffic laws and do not provide a general driver-oriented style of language. For instance, none of these formalisation methods consider a number of important ego vehicle signals (e.g. high/low beam, left/right turn signal) and traffic signals (e.g. traffic lights, stop sign). Moreover, their specifications must be customised for different test scenarios, and lack an automatic method to generate new scenarios as `LawBreaker`'s optimising fuzzing algorithm does.

In this paper, we proposed the STL-based `LawBreaker` to translate traffic laws in a fully decoupled manner, i.e. without any knowledge of the underlying scenario.

7 CONCLUSION AND FUTURE WORK

In this paper, we proposed a framework, `LawBreaker`, for the evaluation of AVs with respect to road traffic laws. A key contribution is its driver-oriented specification language for the description of traffic laws, which is fully decoupled from and compatible with different scenario description DSLs. We proposed a fuzzing engine that searches for different ways of violating the law specifications by maximising a form of specification coverage, i.e. different ways of violating the underlying STL formulas. We implemented and evaluated `LawBreaker` for the state-of-the-art Apollo ADS and LGSVL simulator, and were able to violate 14 Chinese traffic laws, with 173 of the generated test cases causing accidents.

There are several interesting avenues for future work. First, we are interested in finding more efficient methods to generate test cases based on the given specification. Furthermore, we only considered how the ego vehicle should behave in this work, and are interested in exploring how the traffic flow should be when other traffic participants are autonomous vehicles as well.

REFERENCES

[1] 2019. Apollo 5.0. https://github.com/ApolloAuto/apollo/releases/tag/v5.0.0. Online; accessed August 2022.
[2] 2020. Apollo 6.0. https://github.com/ApolloAuto/apollo/releases/tag/v6.0.0. Online; accessed August 2022.
[3] 2021. AVUnit. https://avunit.readthedocs.io/en/latest/. Online; accessed August 2022.
[4] 2022. antlr4. https://github.com/antlr/antlr4. Online; accessed August 2022.
[5] 2022. Autoware.AI. www.autoware.ai/. Online; accessed August 2022.
[6] 2022. LawBreaker: Supplementary Material. https://lawbreaker2022.github.io/. Online; accessed August 2022.
[7] Matthias Althoff, Markus Koschi, and Stefanie Manzinger. 2017. CommonRoad: Composable benchmarks for motion planning on roads. In *Intelligent Vehicles Symposium*. IEEE, 719–726.
[8] Matthias Althoff and Sebastian Lutz. 2018. Automatic Generation of Safety-Critical Test Scenarios for Collision Avoidance of Road Vehicles. In *Intelligent Vehicles Symposium*. IEEE, 1326–1333.
[9] Sai Krishna Bashetty, Heni Ben Amor, and Georgios Fainekos. 2020. DeepCrashTest: Turning Dashcam Videos into Virtual Crash Tests for Automated Driving Systems. In *ICRA*. IEEE, 11353–11360.
[10] Halil Beglerovic, Michael Stolz, and Martin Horn. 2017. Testing of autonomous vehicles using surrogate models and stochastic optimization. In *ITSC*. IEEE, 1–6.
[11] Andrea Censi, Konstantin Slutsky, Tichakorn Wongpiromsarn, Dmitry S. Yershov, Scott Pendleton, James Guo Ming Fu, and Emilio Frazzoli. 2019. Liability, Ethics, and Culture-Aware Behavior Specification using Rulebooks. In *ICRA*. IEEE, 8536–8542.
[12] Chinese Government. 2021. Regulations for the Implementation of the Road Traffic Safety Law of the People's Republic of China. http://www.gov.cn/gongbao/content/2004/content_62772.htm. Online; accessed August 2022.
[13] Anne Collin, Artur Bilka, Scott Pendleton, and Radboud J. Duintjer Tebbens. 2020. Safety of the Intended Driving Behavior Using Rulebooks. In *IV*. IEEE, 136–143.
[14] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. 2017. Robust online monitoring of signal temporal logic. *Formal Methods Syst. Des.* 51, 1 (2017), 5–30.
[15] Wenhao Ding, Mengdi Xu, and Ding Zhao. 2020. CMTS: A Conditional Multiple Trajectory Synthesizer for Generating Safety-Critical Driving Scenarios. In *ICRA*. IEEE, 4314–4321.
[16] Vinayak V Dixit, Sai Chand, and Divya J Nair. 2016. Autonomous vehicles: disengagements, accidents and reaction times. *PLOS ONE* 11, 12 (2016), 1–14.
[17] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *CoRL (Proceedings of Machine Learning Research, Vol. 78)*. PMLR, 1–16.
[18] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. 2019. VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems. In *CAV (1) (LNCS, Vol. 11561)*. Springer, 432–442.
[19] Klemens Esterle, Luis Gressenbuch, and Alois C. Knoll. 2020. Formalizing Traffic Rules for Machine Interpretability. In *CAVS*. IEEE, 1–7.
[20] Georgios E. Fainekos. 2011. Revising temporal logic specifications for motion planning. In *ICRA*. IEEE, 40–45.
[21] Georgios E. Fainekos, Antoine Girard, Hadas Kress-Gazit, and George J. Pappas. 2009. Temporal logic motion planning for dynamic robots. *Autom.* 45, 2 (2009), 343–352.
[22] Georgios E. Fainekos, Hadas Kress-Gazit, and George J. Pappas. 2005. Temporal Logic Motion Planning for Mobile Robots. In *ICRA*. IEEE, 2020–2025.
[23] Francesca M Favarò, Nazanin Nader, Sky O Eurich, Michelle Tripp, and Naresh Varadaraju. 2017. Examining accident reports involving autonomous vehicles in California. *PLOS ONE* 12, 9 (2017), 1–20.
[24] Shuo Feng, Xintao Yan, Haowei Sun, Yiheng Feng, and Henry X Liu. 2021. Intelligent driving intelligence test for autonomous vehicles with naturalistic and adversarial environment. *Nat. Commun.* 12, 1 (2021), 1–14.
[25] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2019. Scenic: a language for scenario specification and scene generation. In *PLDI*. ACM, 63–78.
[26] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Generating effective test cases for self-driving cars from police reports. In *ESEC/SIGSOFT FSE*. ACM, 257–267.
[27] Meng Guo, Karl Henrik Johansson, and Dimos V. Dimarogonas. 2013. Revising motion planning under Linear Temporal Logic specifications in partially known workspaces. In *ICRA*. IEEE, 5025–5032.
[28] Florian Hauer, Tabea Schmidt, Bernd Holzmüller, and Alexander Pretschner. 2019. Did We Test All Scenarios for Automated and Autonomous Driving Systems?. In *ITSC*. IEEE, 2950–2955.
[29] Zhisheng Hu, Shengjian Guo, Zhenyu Zhong, and Kang Li. 2021. Coverage-based Scene Fuzzing for Virtual Autonomous Driving Testing. *CoRR* abs/2106.00873 (2021).
[30] Hardi Hungar, Frank Köster, and Jens Mazzega. 2017. Test specifications for highly automated driving functions: Highway pilot. In *Vehicle Test & Development Symposium*.
[31] Moritz Klischat and Matthias Althoff. 2019. Generating Critical Test Scenarios for Automated Vehicles with Evolutionary Algorithms. In *IV*. IEEE, 2352–2358.
[32] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. 2007. Where's Waldo? Sensor-Based Temporal Logic Motion Planning. In *ICRA*. IEEE, 3116–3121.
[33] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. 2009. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Trans. Robotics* 25, 6 (2009), 1370–1381.
[34] Tanmoy Kundu and Indranil Saha. 2019. Energy-Aware Temporal Logic Motion Planning for Mobile Robots. In *ICRA*. IEEE, 8599–8605.
[35] Morteza Lahijanian, Sean B. Andersson, and Calin Belta. 2012. Temporal Logic Motion Planning and Control With Probabilistic Satisfaction Guarantees. *IEEE Trans. Robotics* 28, 2 (2012), 396–409.
[36] Morteza Lahijanian, Joseph Wasniewski, Sean B. Andersson, and Calin Belta. 2010. Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees. In *ICRA*. IEEE, 3227–3232.
[37] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael B. Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2020. AV-FUZZER: Finding Safety Violations in Autonomous Driving Systems. In *ISSRE*. IEEE, 25–36.
[38] Zhiyu Liu, Jin Dai, Bo Wu, and Hai Lin. 2017. Communication-aware motion planning for multi-agent systems from signal temporal logic specifications. In *ACC*. IEEE, 2516–2521.

[39] Sebastian Maierhofer, Anna-Katharina Rettinger, Eva Charlotte Mayer, and Matthias Althoff. 2020. Formalization of Interstate Traffic Rules in Temporal Logic. In *IV*. IEEE, 752–759.
[40] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *FORMATS/FTRTFT*. LNCS, Vol. 3253. Springer, 152–166.
[41] Satoshi Masuda, Hiroaki Nakamura, and Kohichi Kajitani. 2018. Rule-based searching for collision test cases of autonomous vehicles simulation. *IET Intell. Transp. Syst.* 12, 9 (2018), 1088–1095.
[42] Seyedali Mirjalili. 2019. *Evolutionary Algorithms and Neural Networks - Theory and Applications*. Studies in Computational Intelligence, Vol. 780. Springer.
[43] Galen E. Mullins, Austin G. Dress, Paul G. Stankiewicz, Jordan D. Appler, and Satyandra K. Gupta. 2018. Accelerated Testing and Evaluation of Autonomous Vehicles via Imitation Learning. In *ICRA*. IEEE, 1–7.
[44] Galen E. Mullins, Paul G. Stankiewicz, and Satyandra K. Gupta. 2017. Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles. In *ICRA*. IEEE, 1443–1450.
[45] Christian Neurohr, Lukas Westhofen, Tabea Henning, Thies de Graaff, Eike Möhlmann, and Eckard Böde. 2020. Fundamental Considerations around Scenario-Based Testing for Automated Driving. In *IV*. IEEE, 121–127.
[46] Dejan Nickovic and Tomoya Yamaguchi. 2020. RTAMT: Online Robustness Monitors from STL. In *ATVA (LNCS, Vol. 12302)*. Springer, 564–571.
[47] Philippe Nitsche, Pete Thomas, Rainer Stuetz, and Ruth Welsh. 2017. Pre-crash scenarios at road junctions: A clustering method for car crash data. *Accid. Anal. Prev.* 107 (2017), 137–151.
[48] Jan-Pieter Paardekooper, S Montfort, Jeroen Manders, Jorrit Goos, E de Gelder, O Camp, O Bracquemond, and Gildas Thiolon. 2019. Automatic identification of critical scenarios in a public dataset of 6000 km of public-road driving. In *ESV*.
[49] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. 2019. GeoScenario: An Open DSL for Autonomous Driving Scenario Representation. In *IV*. IEEE, 287–294.
[50] Albert Rizaldi and Matthias Althoff. 2015. Formalising Traffic Rules for Accountability of Autonomous Vehicles. In *ITSC*. IEEE, 1658–1665.
[51] Albert Rizaldi, Jonas Keinholz, Monika Huber, Jochen Feldle, Fabian Immler, Matthias Althoff, Eric Hilgendorf, and Tobias Nipkow. 2017. Formalising and Monitoring Traffic Rules for Autonomous Vehicles in Isabelle/HOL. In *IFM (LNCS, Vol. 10510)*. Springer, 50–66.
[52] Christian Roesener, Felix Fahrenkrog, Axel Uhlig, and Lutz Eckstein. 2016. A scenario-based assessment approach for automated driving by using time series classification of human-driving behaviour. In *ITSC*. IEEE, 1360–1365.
[53] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Martins Mozeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, Eugene Agafonov, Tae Hyung Kim, Eric Sterner, Keunhae Ushiroda, Michael Reyes, Dmitry Zelenkovsky, and Seonman Kim. 2020. LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving. In *ITSC*. IEEE, 1–6.
[54] Yasser Shoukry, Pierluigi Nuzzo, Ayca Balkan, Indranil Saha, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia, George J. Pappas, and Paulo Tabuada. 2017. Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming. In *CDC*. IEEE, 1132–1137.
[55] Cumhur Erkan Tuncali, Georgios Fainekos, Danil V. Prokhorov, Hisahiro Ito, and James Kapinski. 2020. Requirements-Driven Test Generation for Autonomous Vehicles With Machine Learning Components. *IEEE Trans. Intell. Veh.* 5, 2 (2020), 265–280.
[56] Ziwen Wan, Junjie Shen, Jalen Chuang, Xin Xia, Joshua Garcia, Jiaqi Ma, and Qi Alfred Chen. 2022. Too Afraid to Drive: Systematic Discovery of Semantic DoS Vulnerability in Autonomous Driving Planning under Physical-World Attacks. *CoRR* abs/2201.04610 (2022).
[57] Ding Zhao, Henry Lam, Huei Peng, Shan Bao, David J. LeBlanc, Kazutoshi Nobukawa, and Christopher S. Pan. 2017. Accelerated Evaluation of Automated Vehicles Safety in Lane-Change Scenarios Based on Importance Sampling Techniques. *IEEE Trans. Intell. Transp. Syst.* 18, 3 (2017), 595–607.