Generic AuTo-driving Safety checking BeYond collision simulation

ABSTRACT

Safety plays the key role for the success of autonomous driving (auto-driving) vehicles. Due to the expensive cost of physical on-road testing, researchers and engineers have developed simulation-based *virtual testing* methods. However, present practices are still organized in ad-hoc manners and the community yet lacks a generic method for auto-driving safety testing. Toward this objective, we propose a new method Gatsby that provides generic, non-intrusive, and extensible safety checking capabilities for auto-driving virtual testing. Gatsby abstracts the auto-driving simulation into an event-driven computation and provides a way to write extensible safety rules that checks the violations in the simulation data. Gatsby supports multiple types of simulation environments and it has a pluggable design to incorporate new rules in its checking engine. Users can add rules on-demand under this architecture. We evaluate Gatsby with two real-world auto-driving systems and two simulators. Experimental results show that Gatsby confirms one collision and falsifies three collisions from existing works. Gatsby also detects eight new traffic rule violations out of 2,394 test cases with a tolerable cost. Moreover, we design a physical test case derived from Gatsby virtual testing results to test a Tesla FSD vehicle. Our test case successfully make the Tesla vehicle run a red light on the road, showing the effectiveness of the findings of our method.

1 INTRODUCTION

Safety plays the key role for the successful commercialization of autonomous driving (auto-driving) vehicles. However, full road testing of auto-driving vehicle raises unbearable costs. Alternatively, researchers and engineers have leveraged 3D simulators, like SVL [31], CARLA [13], AirSim [28], Udacity [34], etc., to develop cost-effective *virtual testing* methods [9, 15, 24, 25, 30] for evaluating *auto-driving safety*, such as whether the auto-driving system leads to potential traffic rule violations or even dangerous collisions. In general, simulation-based *virtual testing* models the real-world environment into certain abstractions and substitutes the high road testing cost with the affordable costs of computer systems and tester expericences.

A virtual test case usually appears as a simulation configuration that provides static and dynamic settings for the simulator. Testing specialists need to craft hundreds of virtual test cases [7] to assess the vehicle reactions against the simulated sensor data under various scenarios. Recent works have leveraged automated techniques like fuzz testing [18, 19, 26, 38], content reconstruction [16, 17], adversarial learning [4, 10, 14, 22], etc., to produce abundant test cases that drastically ease human efforts.

Despite a variety of efforts in applying virtual testing to auto-driving safety, the present practices are still organized in ad-hoc manners, where one test setup hardly adapts to another test environment. Thus, the community has not yet achieved a generic method to test the auto-driving safety. For example, it is inflexible to port the user-defined driving behavior analysis into other auto-driving systems and simulators. Repeated efforts are needed on engineering with these systems. As a result, The safety checks (traffic rule violations or dangerous activity detection) are hardly portable from one test environment to another. Also, improper simulation data acquisition in virtual testing may introduce imprecision or even faults to the findings [20], due to the misuse of the simulator. Considering that an auto-driving system could be only compatible with a specific simulator, the problem of lacking generic safety test solution becomes more acute.

The problems appeal to new principles that could reduce the ad-hoc testing deficiencies. Toward this objective, we propose a method named Gatsby (**G**eneric **A**u**T**o-driving **S**afety checking **BeY**ond collision simulation), to provide a non-intrusive, generic, and extensible safety checking method in the virtual testing.

Gatsby differs from existing works in threefold endeavors. First, Gatsby adopts a passive approach to obtain the simulation data from the in-between *bridge* middleware. Virtual testing frameworks that use any of the widely-used bridge standards [6, 32, 33] can be seamlessly supported by Gatsby with no intrusive amendments. Second, it abstracts the obtained auto-driving simulation data into event-driven computations. Based on the high-fidelity data, Gatsby transforms the safety checking problem into a general trace analysis problem. Users can replay the events to study and analyze the interested driving behaviors. Moreover, Gatsby has a pluggable design to incorporate multiple safety rules where each rule may correspond to the semantics of a traffic rule. It also provides a extensible way to write rules that users can freely create new rules on-demand to check driving rule violations in simulation.

We have evaluated Gatsby on two auto-driving systems Apollo [8] and Pylot [30], with two simulators SVL [31] and CARLA [13], respectively. Experimental results show that Gatsby confirms one reported safety issues from [19, 26]. It also successfully captures eight new traffic rule violations from 2,300 test cases missed by existing tools. Moreover, Gatsby also falsifies three false positive collision cases of Apollo [8] reported in an recent work [26].

To conclude, we make the following contributions in this work:

- A new generic virtual testing method Gatsby for checking the safety violations in auto-driving simulation.
- The designs of passive data acquisition, event-driven analysis, pluggable rule incoporation, and violation checking.
- The evaluations of two real-world auto-driving systems Apollo and Pylot upon two simulators SVL and CARLA.
- The evaluations of a Tesla FSD vehicle with the physical test cases derived from findings in virtual testing.

We organize the rest of this work as follows. Section 2 reviews background knowledge and states the motivation. Section 3 presents overall architecture, rule formalization, auto-driving behavior abstraction, and the core algorithm. We then perform experiments in

Section 4 and conduct a case study in Section 5. We also report our findings in test Tesla FSD vehicles in Section 6. Finally, we review related work in Section 7 and summarize our work in Section 8.

2 BACKGROUND AND MOTIVATION

In this section, we review the background knowledge, illustrate the ad-hoc deficiencies of present techniques with an example, and briefly discuss our proposed generic approach.

2.1 The Auto-driving System

Figure 1 shows the component level view of a typical auto-driving system. It consists of the software system and a set of hardware sensors and actuators. To avoid ambiguity, we refer an auto-driving system to the software system in this paper.

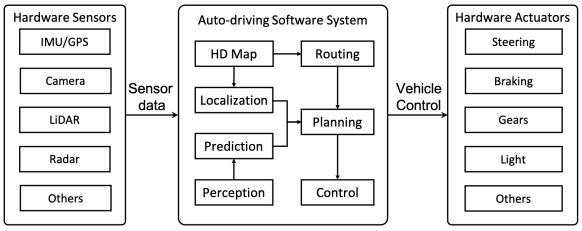


Figure 1: The compnents of an auto-driving system

The left-most hardware sensors in Figure 1 perceive the environment and periodically supply the raw data like image, GPS data, point cloud file and so on to the middle software system. The software system read the senor inputs, estimate vehicle locations, predict traffic situations, and orchestrate together to plan proper control decisions for the right-most actuators. In the perception component, the auto-driving system usually leverages multiple deep neural network models to finish the task of object detection and tracking, traffic sign and signal classification, and lane line detection. Then, after receiving the driving decisions from software control component, the right-most actuators make proper physical signals to the vehicle under control.

2.2 The Auto-driving Virtual Testing

2.2.1 The Typical Setup. Figure 2 shows the typical flow of virtual testing. The 3D simulator first builds a scenario by loading a given test case. The static settings of the test case contain the fixed data in simulation, like the high definition (HD) map, sensors, driving destination, etc. The dynamic part includes parameters that are alterable in a simulation run, such as the non-player character (NPC) cars, traffic signals, light conditions, etc.

After scenario preparation, the simulator launches an auto-driving vehicle in its GUI window, and let the vehicle drive along a route to the destination. Meanwhile, the simulator continuously produces senor data of its virtual environment for the vehicle. In return, the auto-driving system of the vehicle sends the control decisions back to simulator to display the latest vehicle behaviors. This interactive process normally lasts until the user interrupts the simulation.

Note that there also exists a *bridge* that connects the auto-driving system and the simulator. It is a dedicated communication middleware responsible for the actual data exchanging between the bridged systems. We leave more details of it in Section 3.2.

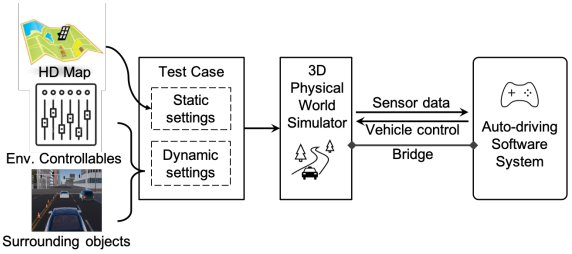


Figure 2: Simulation-based virtual testing for auto-driving

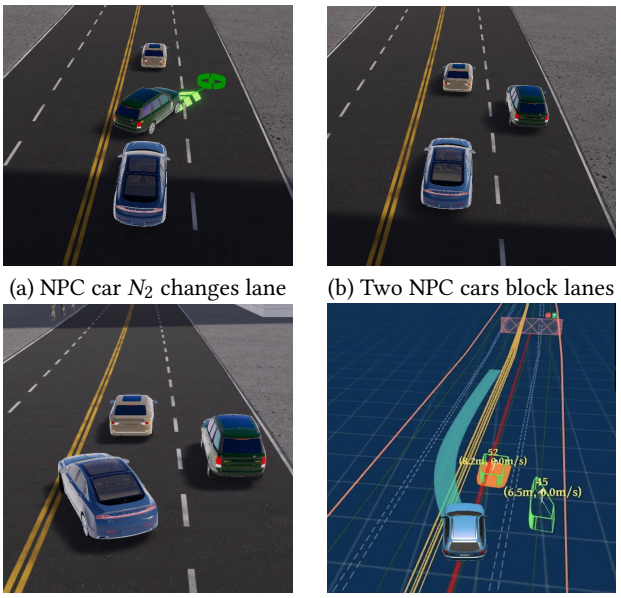
2.2.2 The Common Practices. In Figure 2, we see that the auto-driving vehicle can be viewed as a black-box that runs in a self-contained virtual world. In this world, we testers are able to know the distributions of obstacles and the running auto-driving vehicle. Then, critical risk like collison could be converted to a stateless distance checking problem that checks the distances from an auto-driving vehicle to obstacles. Testers can change the alterable simulation data and measure the distance metric after an auto-driving vehicle reacts to the changes. Moreover, a variety of efforts [18, 19, 26, 27, 38] have integrated automated mutating and advanced searching techniques to seek collision risks of an auto-driving systems, thus forming the common practices of the latest virtual testing. However, we observe that latest techniques are still organized in ad-hoc manners where one test setup can hardly adapt to another test environment. Next, we present an example in Section 2.3 and discuss why latest techniques hardly adapt to the exampled traffic rule violations in Section 2.4.

2.3 A Traffic Rule Violation Example

Figure 3 shows the simulator view and the Dreamview window of a traffic rule violation case we have found in the open source Apollo 5.0 [8]. Apollo is a leading industry-grade auto-driving platform designed for open research study. Dreamview is a GUI software that visualizes the outputs of open source Apollo modules. For more information, please refer to the *official open source site*.

In this simulation scenario, there are three cars on the road including two NPC cars in front and an auto-driving vehicle (ego car) behind, as shown in Figure 3(a). Let us name the lead NPC car as N_1 and the other one as N_2, respectively. N_1 has been stopping on the lane all the time. N_2 is approaching N_1 and decides to change to the right lane, as directed by the arrowed mark in Figure 3(a). After N_2 moves to the right lane, the Apollo-operated ego car is facing N_1, as shown in Figure 3(b). The violation then appears that the ego car drives across the double-yellow line to the lane in the opposite direction, as demonstrated in Figure 3(c).

Figure 3(d) visualizes the intention of this case. At the moment, Apollo's perception module understands through sensor data that two cars are blocking both the current lane and the right lane. It quickly feeds the obstacle information to the planning module for making an immediate driving decision. In such situation, a properly safe decision should be braking the ego car to stop. However, Apollo



(a) NPC car N_2 changes lane (b) Two NPC cars block lanes

(c) Ego car heads wrong lane (d) Planned path in Dreamview

Figure 3: Dreamview and SVL views of a traffic rule violation

5.0 plans a new path to borrow the opposite lane before switching back to the current lane, as shown by the thick curve trajectory.

This case violates certain traffic rules: driving across double-yellow lines and the resulting wrong-way driving. The violations may engender severe consequences if they occur in real-world traffic. The ego car tries not to rear-end the NPC car N_1, but vehicles from the opposite way may strike the ego car head-on. This problem has been fixed in the later version of Apollo open source software.

2.4 Lessons from the Example

2.4.1 Challenges to ad-hoc methods. Automated virtual testing methods often need to run day and night and preserve test cases without human interventions. At the end of testing, testers can investigate the test cases that expose critical consequences. Similar to the automated software testing, the automated auto-driving virtual testing approaches need to set a target to drive the progress, such as the collision- and coverage-driven ways in two state-of-the-art techniques [19, 26].

However, these latest methods would unlikely be aware of the above traffic rule violations in their testing, due to their intrinsic metric-driven principles. For example, the test case in Figure 3 can increase trajectory coverage in ASF [19]. But ASF treat it as a regular test case rather than highlighting it for further investigation since no collision happens. Similarly, AV-Fuzzer [26] would not distinguish the violations since it counts on the distances to obstacles to approach collisions rather than the traffic rules.

Therefore, it is yet challenging to check the generic traffic rule violations even with some latest methods, due to their ad-hoc design manners and different testing tragets. Users may argue that multiple choices can help to accomplish the violation identification, like embedding condition checks into existing tools, simulators, or even auto-driving systems. However, such intrusive amendments require painful efforts in understanding and engineering testing frameworks case by case. Even so, they unlikely work for closed-source simulators or auto-driving systems.

2.4.2 What Kind of Tool We Propose? To check the violations in a generic way, a tool should finish following common tasks.

First, it should capture the dynamic driving behaviors and model the driving progress into an analyzable form. Second, it should define generic forms for the various traffic rules and transform them into checkable specifications in accordance with the above event-driven computation. Third, it should afford a proper way to conduct the checking action and decide the problems.

The simulator builds formatted sensor data for an auto-driving system and visualizes the returned vehicle controls accordingly. Hence, our proposed Gatsby utilizes an essential bridge middleware to capture both sensor data and control decisions to depict the frame-by-frame vehicle behaviors (ref. Section 3.2). From the simulator view, Gatsby generalizes the vehicle interactive driving progress into an analyzable event-driven computation (ref. Section 3.3). After that, Gatsby leverages a rule description language to write generic and extensible traffic rule specifications for checking violations under the event-driven computation (ref. Section 3.4). Finally, Gatsby integrates the above points and checks the potential problems from the simulation data (ref. Section 3.5).

3 METHODOLOGY

In this section, we describe the architecture, data acquisition, key abstractions, rule formalization, and the algorithm of our method.

3.1 The Overall Architecture

Figure 4 shows the overall architecture of our Gatsby method which consists of two layers. The upper layer is consistent with the general virtual testing framework (ref. Figure 2) while the lower layer presents the Gatsby workflow.

Given a test case either from the automated generation or manual effort, the simulator renders a driving scenario and launches an auto-driving vehicle in this visual scenario. Then, the simulated vehicle starts to drive on a route computed by the auto-driving system behind it. During this phase, the simulator continuously constructs virtual sensor data and feeds it to the running vehicle through the bridge. The auto-driving system adjusts the vehicle driving state based on the sensor inputs, and sends its control directives back to the simulator to update the visual vehicle maneuvers.

During the simulation, Gatsby also listens the *Bridge* to acquire the bi-directional data (ref. Section 3.2). The data then goes to the *Simulation Log Data Interpreter* which distinguishes the data standard and deserialize the data into analyzable forms. The interpreted data then flows into the Gatsby *Engine* for further usage.

A pluggable rule (ref. Section 3.4) is a user-defined textual specification that describes the semantics of a traffic rule. It states the expected regulatory behavior that the auto-driving vehicle should comply. Users can write multiple rules and plug them into Gatsby to check whether the driving states of an auto-driving vehicle always comply with the rules.

$$alert(ego.speed \leq (lane.speedLimit + 10)) \quad (1)$$

For instance, a simple rule like Rule (1) checks if the speed of an auto-driving vehicle (*ego.speed*) ever exceeds the speed limit for

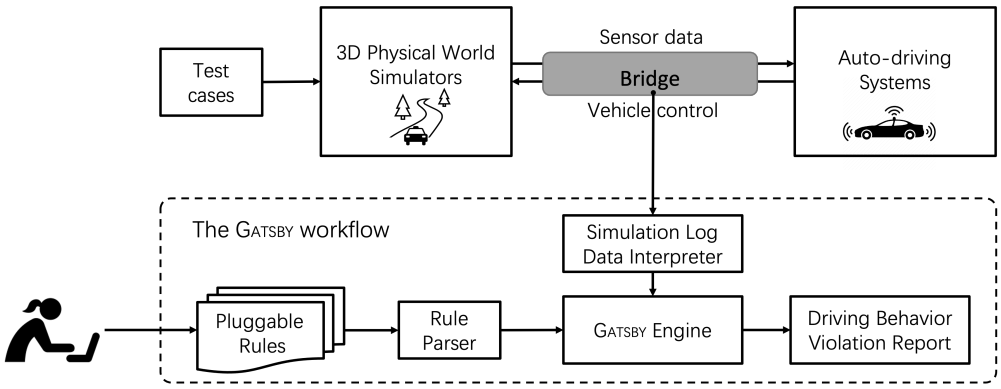


Figure 4: The overall architecture of our auto-driving safety checking tool Gatsby

more than 10 miles per hour. With a supplied rule, Gatsby uses a *Rule Parser* to parse the textual rule into a syntax tree and feeds the parsed rule to the Gatsby *Engine* for further processing.

The Gatsby *Engine* owns the core functionality that manages the rules, generates the checkable rules, re-constructs the driving states upon the simulation data, and performs the rule violation checking. We leave more details in Section 3.5.

After judging whether the simulated driving behaviors conform with the specified rules in Gatsby engine, we levergae a component to generate the violation report. The report includes sets of events that violate different rules to pinpoint the exact simulation data frames triggering the violations.

3.2 Data Acquisition through the Bridge

3.2.1 The role of the bridge. In practice, simulation-based auto-driving virtual testing often needs a dedicated communication middleware that ensures the two-way data exchanging.

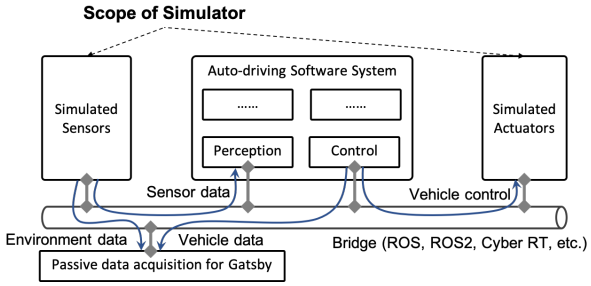


Figure 5: The bridge

We call such middleware the *bridge*. Figure 5 shows the role of the *bridge* that offers the infrastructural communication support for the connected simulator and auto-driving system.

Conceptually, a simulator virtualizes the hardware sensor to send data to the auto-driving software, and the latter outputs control directives back to simulator through the simulated actuators. From the realization perspective, this two-way communication works through the *bridge*, as shown at the bottom of Figure 5. The *bridge* provides loose coupling publisher-subscriber interfaces for all the data exchanging tasks. The exchanged data is ogranized in widely used formats (e.g., ROS [32], ROS2 [33], Cyber RT [6], etc) in auto-driving simulation. Periodically, runtime sensor data (e.g, LiDAR, Camera, and GPS) enters *bridge* through agreed interfaces called topics. The auto-driving perception module who subscribes needed *bridge* topics then persistently receives the data to drive subsequent pipelined computation. Likewise, the simulator that subscribes vehicle control topics retrieves quantitative directives like the steering, throttle, and brake to update the vehicle status in its GUI window.

3.2.2 Data acquisition and processing. The *bridge* architecture makes the **passive data acquisition** of bi-directional data be feasible. Applications can subscribe a list of data topics released on *bridge* without affecting the running simulation. Also, they can stream data into persistent files and aggregate the obtained data into ordered snapshots of sensor input and the counterpart vehicle state. Owning to the holographic features of the aggregated data, we can conduct further analysis upon the behaviors of an auto-driving vehicle against the virtualized sensor inputs.

In auto-driving simulation, Gatsby passively acquires the simulation as described above. It obtains the vehicle data (location, direction, velocity, speed, etc) and the environment data of the surroundings (obstacles, traffic signs, lane lines, etc.) from the *bridge*. Then, Gatsby distinguishes the data standard to first parses the vehicle chassis data. Following the timing cardinal of the chassis data frames, it aligns chassis data with the parsed correlated environment data, to construct a sequence of events to represent the simulation progress. Note that here we assume the integrity of the obtained simulation data, and focus on the behavior analysis of the auto-driving simulation. After the data acquisition and subsequent abstractions (ref. Section 3.3), Gatsby performs checks (ref. Section 3.5) with the data events and the supplied rules to detect violations in simulation.

3.3 Auto-driving Behavior Abstraction

Transforming the simulated auto-driving vehicle dynamics into an analyzable problem raises challenges in understanding the system and the environment. Therefore, we need multiple abstractions in Gatsby for modeling the needed information.

- The system abstraction that models the continuous simulation of an auto-driving system into an analyzable system.

- The input abstraction that models the external data into the formatted inputs to the abstracted system.
- The rule abstraction that models the conceptual traffic rules into checkable forms in line with the system and the data.

3.3.1 The system abstraction. The simulator affords a stable and repeatable virtual world for testing auto-driving vehicles. We obtain an observation from the simulation runs. That is, assuming the virtual sensors always work normally, the execution of an auto-driving system could be abstracted as a stateful event-driven computing machine that periodically receives the event-like sensor input. It then computes a new system state upon the event and its current state, and outputs control decisions that reflect its observable system state. In this way, we establish the system abstraction as a high-level data-driven computing machine.

3.3.2 The input abstraction. Next, observing that the simulation runs frame by frame, we formulate the environment data in simulation scene at the beginning of each frame as a trace of inputs to an auto-driving system . Then, the continous environment data in simulation can be modeled into an ordered trace of inputs in chronological order. An input contains the structured raw data for the sensors. Accordingly, the auto-driving vehicle reactions upon each input can be formed as a trace of observable system states. Given the two traces, we align them in terms of the vehicle chassis state time-series, merge each input and the corresponding vehicle state into an event, and construct a sequence of events. With this abstraction, we can leverage the above abstract event-driven computing system to support the step-by-step analysis of sensor input and the corresponding driving behavior together. As a result, we finish the input abstraction and transfer the auto-driving behavior analysis into a data-driven trace analysis.

3.3.3 The traffic rule abstraction. Further, we need to model a traffic rule into an checkable form. Though people can find human-readable traffic rules in the drivers' handbooks, there are no uniform ways that incorporate the multifarious rules into auto-driving virtual testing. In general, one needs to first map the conceptual traffic rules into a computer-understandable form. Then, the computer system transfers the form into checkable specifications that state the expected behaviors. Further, violations of the specifications upon some given driving data may expose the improper driving logics contained in that data.

We may have multiple choices in abstracting the conceptual rules into computer-understandable forms. For example, the networked timed automata [5] with integer and array types can work with Uppaal [21] to check the real-time systems properties. However, we expect the abstraction should accommodate following points:

- Be general and easy for the new rule incorprations.
- Be independent of specific auto-driving systems.
- Be suitable for the automated reasoning in Gatsby.

Under the above event-driven computing schema and the data abstraction, we propose a light-weight rule description language to build stateless and stateful rules for checking purpose. We leave its formalization in Section 3.4. In summary, we design a set of auto-driving related primitives and a set of operators while users can freely assembly them to model various rules.

Rule	r	::=	$\texttt{alert}(b)$
		|	$r_1; r_2$
Bool	b	::=	$c_1 \vee \cdots \vee c_n$
Conjunct	c	::=	$e_1 \wedge \cdots \wedge e_n$
Expressions	e	::=	$p \mid n \mid n.n \mid F(e_1, \cdots, e_k)$
		|	$e_1\ aop\ e_2$
		|	$e_1\ lop\ e_2$
		|	$true$
		|	$false$
Predefined APIs	F	::=	$f_1 \mid f_2 \mid \cdots \mid f_n$
Predefined Primitives	P	::=	$p_1 \mid p_2 \mid \cdots \mid p_n$
Arithmetic Operators	aop	::=	$+ \mid - \mid * \mid /$
Logic Operators	lop	::=	$> \mid \geq \mid < \mid \leq \mid ==$
		|	$\neq \mid \&\& \mid \mid\mid \mid ! \mid ! <$
Types	τ	::=	$\texttt{Boolean} \mid \texttt{Int} \mid \texttt{Double}$
Values	v	::=	$true \mid false \mid n \mid n.n$

Figure 6: Language Syntax of $\mathcal{L}$.

3.4 The Traffic Rule Formalization

In this section, we formalize the traffic rule supported by Gatsby, using a formal language $\mathcal{L}$ to present the rule syntax for rule generation and violation checking in Gatsby.

To clarify the scope of our work, we emphasize that the following formalization aims to demonstrate the automated checking capability of of our tool. We do not intend to cover all kinds of real-world traffic rules by ourselves. The fulfillment of this objective requires much domain knowledge and significant human labors, which exceeds the scope of this work. Instead, we try to provide a rich set of primitives that are closely related to the features of auto-driving simulation data. Community users can leverage the primitives to assembly their own new rules to test various behaviors of the auto-driving systems supported by our framework.

Figure 6 shows the syntax of $\mathcal{L}$. A rule r usually consists of the disjunctive form of multiple conjunct expressions. An expression can be composed of a predefined primitive, an integer, a floating point number, a predfined API function, and multiple sub-expressions connected by the arithmetic or logic operators.

We design a rich set of predefined primitive to provide the information of the road, the auto-driving vehicle, the obstacles, etc. Users can quickly write new rules like stacking Lego blocks.

$$alert((ReachStopSign(ego.location, 2) \\ \&\& \ ego.speed > 5.0) == false) \quad (2)$$

For example, the expression in Rule (2) owns two sub-expressions connected by "==". The first sub-expression has a function *ReachStopLine* and two primitives *ego.location* and *ego.speed*. Function *ReachStopSign* checks if an auto-driving vehicle (*ego.location*) approaches a stop sign in 2 meters. In addition, the first sub-expression also checks whether the vehicle speed (*ego.speed*) exceeds 5 mph. If the vehicle is approaching a stop sign at a higher speed under successive events of simulation data, then the first sub-expression may evaluate to *true* but the overall expression becomes *false*. As a result, a violation of Rule (2) happens in those events.

For simplicity, $\mathcal{L}$ also treats the predefined APIs F as built-in primitives rather than explicit declarations. F encapusulate reusable computations to ease the effort of rule writing. Thus, we can easily embed functions like the above *ReachStopLine* into a rule.

3.5 The Gatsby Engine Internal Structure

In this section, we present the internal structure of the Gatsby engine in Figure 7 and the corresponding algorithm in Algorithm 1.

In Figure 4, a rule parser parses the textual rules written in $\mathcal{L}$ syntax (ref. Section 3.4) and feeds the parsed syntax tree to the Gatsby engine. Also, the simulation log data goes through the interpreter and then the processed data flows into the Gatsby engine. In Figure 7, we illustrate the internal engine architecture.

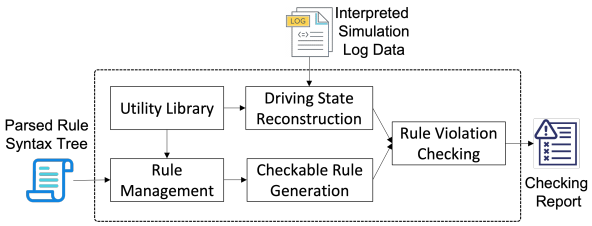


Figure 7: The internal architecture of the Gatsby engine

Specifically, the *RuleManagement* loads the rule syntax tree with the help of the *UtilityLibrary*. It also traverses the tree to obtain the primitives and functions from the loaded rule. However, at this point, the rule is still not ready for checking. The subsequent *CheckableRuleGeneration* then maps the obtained primitives and functions to certian data interfaces to instantiate the rule into the checkable form. On the other side, the *DrivingStateReconstruction* component deserializes the interpreted simulation log data in time-series order and constructs a sequence of events to model the framed simulation execution states. After that, Gatsby iterates over the data events and checks the rules one-by-one against the data contained in each event, thus forming an event-driven analysis schema.

We use Algorithm 1 to accommodate the processing flow in Figure 7 for detailed explanations. Algorithm 1 consists of the main procedure ExecGatsby and three auxiliary functions in corporation with ExecGatsby on processing the supplied *data* and *rule*.

Initially, ExecGatsby starts with the interpreted *data* and a parsed *rule* syntax tree. The *data* from the communication bridge includes various topics of simulator senor data and the control data from auto-driving system to simulator. At line 3, we call *DrvingStateReconstruction*. It first extracts the chassis data from *data*, and calibrates *data* in terms of the timestamps of the chassis data,

Algorithm 1 The main algorithm of Gatsby

```
1:  Initially: Start ExecGatsby with the interpreted data from
    simulation bridge and a parsed rule from the parser.
2:  procedure ExecGatsby (data, rule)
3:      events ← DrivingStateReconstruction (data);
4:      rule′ ← RuleManagement (rule);
5:      θ ← CheckableRuleGeneration (rule′);
6:      for e in events do
7:          for r in θ do
8:              violated ← RuleViolationChecking (e, r);
9:              if violated then
10:                 Record violated, e.id, and r;
11:             end if
12:         end for
13:     end for
14: end procedure
15:
16: function DrivingStateReconstruction (data)
17:     Let δ ∈ data be the chassis data of the simulated vehicle;
18:     Perform calibration of data according to δ.timestamp;
19:     event ← convert data to a sequence of framed events;
20:     return events;
21: end function
22: function RuleManagement (rule)
23:     Load rule and check the rule integrity;
24:     rule′ ← Traverse rule syntax tree and annotate primitives;
25:     return rule′;
26: end function
27: function CheckableRuleGeneration (rule′)
28:     Let θ be the global rule list maintained in Gatsby;
29:     rule′′ ← map primitives of rule′ to data interfaces;
30:     θ.append(rule′′);
31:     return θ;
32: end function
33: function RuleViolationChecking (e, rule)
34:     Substitute data interface in rule with the event data;
35:     result ← perform rule evaluation;
36:     return result;
37: end function
```

thus forming a time-series data flow (lines 17-18). Then, this function frames different data topics with relevant timestamps, such as speed, position, obstacles, traffic signals, etc, into an event and converts the data flow to a sequence of events (lines 19-20). Here each event represents a snapshot of the simulation execution state.

After processing *data*, we call *RuleManagement* (line 4) to load the *rule* syntax tree for integrity check (line 23). Then, we traverse the syntax tree and annotate the primitives to form *rule′* (lines 24-25). For brevity we treat the predefined API functions as primitives as well. The annotated *rule′* then goes into *CheckableRuleGeneration* (line 5) which maps the primitives to corresponding data interfaces and appends the mapped *rule′′* to the global rule list (lines 29-30).

Next, we iterate the *events* and perform the checking, as shown between line 6 and line 13. To be specific, for each event *e*, we iterate the global rule list θ to fetch the rules and check them against *e* in *RuleViolationChecking* (line 8). Function *RuleViolationChecking*

substitutes the embedded data interface in the input *rule* with the data contained in *event* to instantiates the rule for evaluation and returns the result (lines 34-36). If the returned *violated* is *true*, we observe a violation of the current rule and record it (lines 9-10). The main procedure repeats execution until all the *events* are checked. Then, we can collect the reported violations for further analysis.

4 EVALUATION

4.1 Research Questions

We position Gatsby as a safety checking technique that enriches the virtual testing family in a synergistic way. To evaluate the effectiveness and the overhead, we make following research questions:

RQ1 Can Gatsby identify or falsify the collisions reported by existing virtual safety testing tools?

RQ2 Can Gatsby detect new traffic rule violations by checking the simulation data of test cases from existing tools?

RQ3 Is Gatsby a cost-effective testing approach that has tolerable overhead compared to existing tools?

4.2 Experimental Settings

We have implemented Gatsby for the Cyber RT and the ROS/ROS2 bridges and divide the pipeline of Gatsby into serveal steps.

We evaluate Gatsby upon two auto-driving systems Apollo v5.0 and Pylot v0.32. They work on two simulators SVL 2021.2.1 and CARLA 0.9.10, respectively. Apollo is a mature industry-class open source auto-driving system which has been actively evolving since the v1.0 release in July 2017. It uses the Cyber RT bridge instead of ROS bridge after v3.5. Pylot is an auto-driving platform for developing and testing the core components of auto-driving that works with CARLA and the physical cars. Pylot has a streaming component upon ROS2. Thus, we obtain the bridge data with ROS2 utilities. We also tried another system Autoware.ai. However, we cannot properly witness its traffic light detection results even following the official documents. After searching the issue, we found that it has been a common problem that still lacks stable solutions. Thus, for the fair comparison concern, we did not evaluate Autoware.ai. However, since Autoware.ai also builds on ROS, we deem that our approach should have no technical barriers for it.

All the experiments were performed on a machine running Ubuntu 18.04 64-bit Server Linux with Intel(R) Xeon(R) 2.20GHz CPUs 48 cores and a 512GB RAM. The simulation of each scenario with existing fuzzing tools is allowed to run at most 12 hours.

4.3 Checking for Known Collision Cases

We answer the first question in this section. To assess the effectiveness of Gatsby, one natural question is that whether Gatsby can also identify the auto-driving collisions found by exisiting techniques. That is, if Gatsby has reasonable results upon the known problems, we can gain certain confidence and proceed further evaluation, like trying out a bug detector against known CVEs.

With the purpose, we investigate two state-of-the-art fuzzing tools in virtual testing, i.e., AV-Fuzzer [26] and ASF [19], and leverage their reported collisions as the obstacle to grade the capability of Gatsby. Both two fuzzers can work with the Apollo 5.0 system.

We place the settings and the results of this study in Table 1. The first column indicates the reported incidents from AV-Fuzzer and ASF all happen under the `Lane follow` scenario where the auto-driving vehicle and the NPC cars drive on a straight road.

Note that AV-Fuzzer and ASF can automatically generate handreds of similar scenes to enforce the subtle environment changes against the Apollo 5.0 system. We in total collect four representative collision scenes from running the two tools, as shown in the second column by distinguishable names. The term *scene* represents a specific simulation configuration under the parent scenario. One scene usually corresponds to one simulation test case.

Table 1: Gatsby checks the collisions cases reported by [19, 26].

	Scene	NPCs	AV-Fuzzer	ASF	Gatsby – collision	
					Objective	Perception
Lane follow	hit_adv_npc	1 moving	N/A	✔	✔	✔
	side_hit_npc	1 stopped	N/A	✔	✗	✔
	rear_end_npc	2 moving	✔	✔	✗	✔
	ego_hit_cone	1 moving, 1 stopped	N/A	✔	✗	✔

(a) hit_adv_npc (b) side_hit_npc (c) rear_end_npc (d) ego_hit_cone

Figure 8: Visualizations of simulated collisions in Table 1.

The third column lists the information of the NPC cars. We tried diverse combinations of moving and stopped NPCs in the scenes to form varied traffic situations. The fourth and fifth columns show where the collision scenes come from. N/A means the scene is not applicable to a fuzzer. By contrast, ✔ means the scene has a collision under the search of a fuzzer. In our evaluation, we were unable to reproduce the original five rear-end collisions in AV-Fuzzer due to the obsolete simulator LGSVL 2019.05. Alternatively, we experimented AV-Fuzzer with SVL 2021.2.1 and Apollo 5.0. With this new setting, AV-Fuzzer still detected a real-end collision. In addition, ASF found three more collision cases, as shown in Figure 8. It is worth mentioning that finding collisions in Apollo simulation is challenging due to the mature status of this industry-grade system. Thus, the limited collisions are indeed valuable exceptions.

The last two columns under **Gatsby-Collision** represents the checking results of Gatsby. One unique feature of Gatsby is that it supports running with the **Objective** data, which is the original virtual senor data synthesized by the simulator. With this data, Gatsby checks whether an auto-driving system reacts correctly to the actual environment data as a whole. Meanwhile, Gatsby can also run with the **Perception** data, which is the processed result of the **Objective** data by the perception component of an auto-driving system. Due to the potentially problematic output of the used DNN models, the perceived information may differ from the actual information in the **Objective** data. Running Gatsby with **Perception** data can help check whether the subsequent components in an auto-driving system still perform well if a perception error happens. The **Perception** data is also available on bridge since simulator always needs to display the perception result.

More importantly, if the checking result of **Objective** data is contraidictory with that of **Perception** data, there should be a problem either in the simulator usage or in the auto-driving system.

We observe that Gatsby reports all ✔ in the **Perception** column. This means Gatsby has found collisions with the **Perception** data, which confirms the results of existing fuzzers. However, the three ✗ in the **Objective** column indicate that no collisions appear in those scenes when running with the simulator **Objective** data.

We studied the results and found that collision cases (b), (c), and (d) in Figure 8 are false positive cases that should not happen in real-world road testing. But why they happen in simulation? The answer stems from the usage of the simulator in the evaluated fuzzing tools. To approach collision in each simulation epoch, fuzzers pause a running simulation, obtain the present traffic, compute the adversarial NPC maneuvers, and finally resume simulation to enforce the NPC behaviors. However, the pause action only suspends the simulation world while the auto-driving system keeps rolling. Such short out-of-sync status accumulates and finally results in the collisions in two fuzzers. By contrast, Gatsby falsifies that the collisions should not happen. Thus, Gatsby offers a two-tier check for the reported driving issues with the **Objective** and **Perception** data.

Answer to RQ1: Gatsby can find the known collision cases reported by existing tools. Moreover, it also falsifies multiple simulation collisions that cannot be distinguished by state-of-the-art fuzzing-based virtual testing tools.

4.4 Checking the Traffic Rule Violations

In this section, we answer the second research question by applying Gatsby to the simulation data obtained from running all the test cases from existing fuzzers. Our purpose is to experiment whether Gatsby can find new traffic rule violations missed in fuzzing.

We first state the targeted violations in this experiment:

(1) Wrong-side driving: an auto-driving vehicle drives against the direction of the traffic on the lane.
(2) Crossing double-yellow line: an auto-driving vehicle strides over solid double-yellow parallel lines.
(3) Stop sign violation: an auto-driving vehicle passes the stop sign and drives into the intersection without stopping or following the right order to go.
(4) Run a red light: an auto-driving vehicle passes the stop line and drives into the intersection at a red traffic light signal.

Table 2 shows the result by running Gatsby with the bridge data. The first column lists three scenerios and the second column lists the scenes under a scenario. For example, the `Stop sign` scenario has the `two_way` and `four_way` stop sign scenes while the `Traffic light` scenario has three scenes as `protected_left_turn`, `protected_right_turn` and `straight_driving`. The next column shows the obstacle information as how the NPC cars act in each scene. For instance, the block-go NPC case under the `Traffic light` scenario means a NPC blocks the intersection and leaves after a while.

The rest columns show the traffic rule violation results under two fuzzing methods [19, 26] (**Fuzzing**), our Gatsby method with simulator synthesized sensor data (**Gatsby-Objective**), and Gatsby with the perceived data (**Gatsby-Perception**), respectively. In the following, we use **GO** and **GP** to denote them for short.

Table 2: The results of traffic rule violation checking in Gatsby.

Scenarios	Scene	Obstacle	Fuzzing		Gatsby–Objective		Gatsby–Perception	
			Apollo	Pylot	Apollo	Pylot	Apollo	Pylot
Lane follow	lane_borrow	two NPCs	0	0	(1), (2)	0	(1), (2)	0
Stop sign	two_way	stop-go NPC	0	0	0	0	0	0
		transverse NPC	0	0	0	0	0	0
	four_way	stop-go NPC	0	0	0	0	0	0
		transverse NPC	0	0	0	0	0	0
Traffic light	straight _driving	block-go NPC	0	0	(4)	(4)	(4)	(4)
		no NPC	0	0	(4)	(4)	(4)	(4)
	protected_ left_turn	block-go NPC	0	0	(4)	(4)	N/A	N/A
		no NPC	0	0	(4)	(4)	N/A	N/A
	proteced_ right_turn	block-go NPC	0	0	(4)	(4)	N/A	N/A
		no NPC	0	0	(4)	(4)	N/A	N/A

Not surprisingly, the fuzzing methods detect `0` traffic rule violations since they primarily target simulation collisions, as shown in the two columns under **Fuzzing**. Then, we look into the Gatsby results. Overall, the **GO** and **GP** have similar results, indicating a consensus in their detections. We use the form (N) to denote the found violations where N∈[1,4] regarding the provided four rules. Still, `0` means no violations and N/A means non-applicable results.

In the `lane_borrow` scene, Gatsby found 2 violations in Apollo in terms of Rules (1) and (2), which were explained in Section 2.3. We analyzed Apollo 5.0 code for these violations. The root reason stems from a missed check of the double-yellow line lane boundary type when the auto-driving vehicle tries to borrow a wrong-direction lane. This issue had already been confirmed and fixed by the Apollo developer team. In Pylot, Gatsby did not find such problem.

Under the `Stop sign` scenario, Gatsby found no violations in both Apollo and Pylot. However, Gatsby reported violation (4) in both systems in all three `Traffic light` scenes with the **GO** setting. Gatsby reported violation (4) of the two systems in the `straight_driving` scene with **GP**. After analysing the source code, we found that the violations came from the realizations of an implicit state machine, whose problematic transition made the auto-driving vehicle run-red-light. We make a detailed study in Section 5.

Answer to RQ2: Gatsby can effectively identify several traffic rule violations due to in the auto-driving systems, by analyzing the simulation data of the test scenes generated by existing tools.

4.5 Computational Overhead of Gatsby

In this section, we answer the third question by evaluating the computational overhead of Gatsby in above experiments.

Table 3: The execution cost statistics of Gatsby.

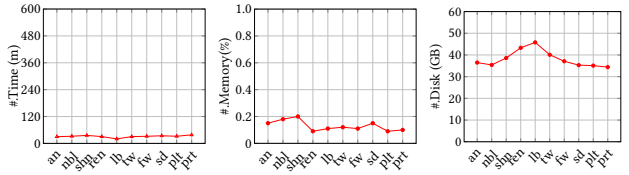
Scenarios	Scene	Fuzzing		Gatsby – Cost		
		#.Time (m)	#.Tests	#.Disk (GB)	#.Memory(%)	#.Time (m)
Lane follow	adv_npc	720	236	20.4	0.15	36.5
	npc_block_lane	720	288	22.4	0.18	35.4
	side_hit_npc	720	244	20.0	0.20	38.6
	rear_end_npc	720	278	21.7	0.09	43.3
	lane_borrow	720	283	22.6	0.11	45.8
Stop sign	two_way	720	210	16.1	0.12	40.1
	four_way	720	201	16.0	0.11	37.1
Traffic light	straight_driving	720	208	17.3	0.15	35.3
	protected_left_turn	720	210	17.0	0.09	35.1
	proteced_right_turn	720	215	17.6	0.10	34.4

We present the cost statistics in Table 3. The first two columns list the driving scenarios and scenes in above evaluations. Column three shows the time used in generating the test cases by fuzzing and column four gives the amount of test cases within the given time. The rest columns under **Gatsby-Cost** show the cost of Gatsby in three dimensions, including the total execution time (**#.Time (m)**), the ratio of used memory against the total memory (**#.Memory (%)**), and the disk for data storage in gigabyte (**#.Disk (GB)**).

As set in Section 4.2, the simulation in existing fuzzing tools runs at most 12 hours. Hence, each cell in column three of Table 3 has

the same value of 720 (m). In column four, the amount of generated test cases ranges from 201 to 288. Unlike software fuzzing where fuzzers may generate thousands of test cases in several hours, the simulation-based auto-driving fuzzing runs much slower. Replaying the interested tests for error diagnosis also requires long time budget. By comparison, GATSBY can finish fast analysis of each scene tests within 45 minutes, as shown in the last column of Table 3.

Moreover, the amount of used memory by GATSBY remains stationary regardless of the driving scenes, where the average consumption is less than 0.2% of the total physical memory, as shown in column six of Table 3. Column five depicts the disk consumption of the bridge data storage for GATSBY. We observe that the cost of GATSBY mostly comes from the heavy disk consumption. The simulation scene `lane_borrow` requires the most data storage of 45.8 GB where `protected_right_turn` has the minium requirement of 34.4 GB. However, considering the decreasing price of the hard disks, we regard such cost to be tolerable.



(a) The time (b) The memory (c) The storage

Figure 9: The time, memory, and storage cost of GATSBY.

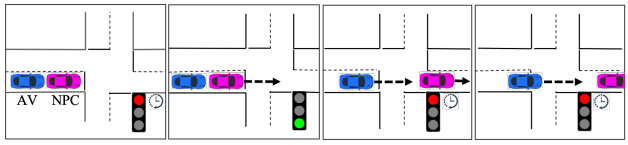
Further, we draw Figure 9 to visualize the cost of GATSBY. The *x-axis* of each subfigure lists the abbreviations of scenes in Table 3 and the *y-axis* shows each cost statistics. We observe that despite the fluctuant storage demands for various driving scenes, GATSBY has stable time cost and extremely low memory consumption. Thus, it is a cost-effective virtual testing technique.

Answer to RQ3: GATSBY consumes tolerable resources in execution time, used memory, and the storage for simulation scene data. It is a cost-effective checking method that uses extremely low memory and produce fast results.

5 CASE STUDY

In this section, we study two safety violation case.

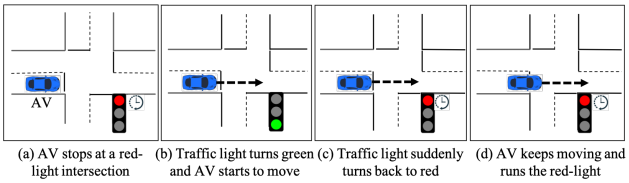
5.1 Run-a-red-light after Light Signal Change



(a) AV and NPC stop at a red-light intersection (b) Traffic light turns green, NPC and AV start to move (c) Traffic light turns red before AV passes the stop line (d) AV keeps moving and runs the red-light

Figure 10: Run-a-red-light with a NPC car ahead

Figure 10 illustrates the progress of a run-a-red-light traffic rule violation caused by an auto-driving vehicle (AV). Initially, the AV and a front NPC car reach the intersection, as shown in Figure 10(a). At this time, the traffic light is red and both vehicles stop before the stop line as expected. Then, in Figure 10(b), the traffic light turns green. So both AV and NPC car start to move. In Figure 10(c), the NPC car enters the intersection and goes over the traffic light. However, the light shortly turns from green to red, right before the AV reaches the stop line. At this point, the AV is expected to stop since it has not passed the stop line yet. However, in Figure 10(d), the AV keeps moving rather than braking to stop. Thus, it inevitably runs the red light and causes a traffic rule violation.



(a) AV stops at a red-light intersection (b) Traffic light turns green and AV starts to move (c) Traffic light suddenly turns back to red (d) AV keeps moving and runs the red-light

Figure 11: Run-a-red-light without front NPC car

It seems the NPC car may impact the behavior of the AV in Figure 10. To eliminate the side effect, we re-conduct the scenario without any NPC participant that blocks the lane, as shown in Figure 11.

This time, in Figure 11(a), the AV first stops before the intersection due to the red light. Then, in Figure 11(b), the traffic light turns green and the AV begins to move as usual. However, in Figure 11(c), we make the light suddenly switches back to red. At the moment, the AV just reaches the stop line. It is expected to brake and stop because of the red light. Unfortunately, the AV fails to stop but continues driving forward. Hence, it runs the red light again.

5.2 The Irreversible State Transition Problem

The situations in Figure 10 and Figure 11 may arise from various ways. For example, an obstacle could block the intersection a while until the signal changes from green to red. Also, a perception error in auto-driving system could mis-detects a frame of red light and form the `red-green-red` case in Figure 11. In this case, the traffic signal remains red all the time but the vehicle makes the wrong decision. In either way, it indicates that the auto-driving system has certain fault under the traffic light scenario.

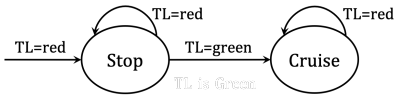


Figure 12: The implicit and problematic transition model

After the investigation, we present the reason of the violation by a simplified state machine in Figure 12. In the systems, when AV reaches an intersection at a `red` traffic light (TL=red), it would enter the `Stop` state. The AV then periodically detects the traffic light and stay in `Stop` if TL remains `red`. Once AV perceives a `green` light (TL=green), it transits to `Cruise` and starts moving right away.

The problem is that the transition from `Stop` to `Cruise` is one-way. Once the irreversible transition happens due to a one-time perception error or a malicious traffic light attack, the vehicle cannot switch back from `Cruise` to `Stop`. Instead, it stays in `Cruise` state and keeps moving, despite it perceives the actual `red` signal.

5.3 How GATSBY Detects the Problem

Since no collision happenes in the above two cases, existing collision- or coverage-driven testing techniques unlikely report such silent violations. By contrast, GATSBY would not miss them.

For the run-red-light violation problem, users with different focuses may write various versions of rules to inspect the problem. We list our rule that finds the problem as follows:

$$alert((PassStopLine(ego.location, ego.length) \\ \&\& \ trafficlight.color == red) == false) \quad (3)$$

Rule (3) means that if an auto-driving vehicle (ego car) starts passing a lane stop line then the corresponding traffic light color should not be *red*. The function *PassStopLine* returns *true* if the auto-driving vehicle's location (*ego.location*) passes the lane stop line in less than *ego.length* meters. Otherwise, it returns *false*.

Argument *ego.length* indicates this rule takes effect until the ego car fully passes the line. Testers can freely alter this argument to extend or shrink the checking window and our choice can capture enough events as the evidence if the problem really happens.

Going back to Figure 10(d), the blue ego car begins to pass the intersection stop line when the front traffic light color has been red. Undoubtedly, Gatsby detects that this behavior violates Rule (3). Similarly, in Figure 11(d), the blue ego car reaches the lane stop line and continues driving, while the traffic light has turned red. The data events of this behavior also make Rule (3) evaluate to be *false*, thus Gatsby identifies this run-red-light problem.

Note that Rule (3) can capture most cases but may output false positives under corner cases. In such situation, we can add more predicates to refine Rule (3) or relate multiple events in rule to perform a finer-grained checking. Also, Gatsby can detect safety violations and pinpoint the problematic data. However, it has no knowledge about which system code logic introduces the detected problems. Testers need to further leverage the error-triggering data to debug and improve the system.

6 TESTING TESLA FSD WITH OUR FINDINGS

Despite Gatsby found several problems in virtual testing, we often have a question: would the problems likely happen on the road? To answer the question, we utilize Figure 10 to build a physical test case for testing a Tesla FSD (Full Self-Driving) vehicle.

We choose Tesla for three reasons: (1) Apollo and Pylot need certain hardware, dedicated vehicle, and proper installment. (2) Tesla is the most popular intelligent vehicle and we can test it as a black-box. (3) If a third-party Tesla also expose similar issue, then the syndrome might be a general safety issue.

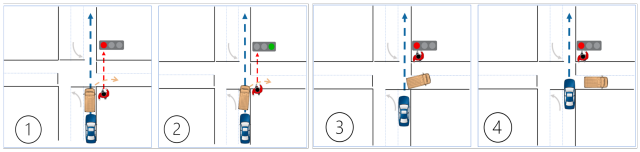


Figure 13: Four steps of the Tesla test case

We used a 2020 Tesla Model *Y* with latest FSD 10.5 (on Nov 29 2021) as the testbed. Initially, *Y* run in FSD mode on a pre-selected route that passes an intersection with traffic lights. Another vehicle *X* runs ahead of Tesla Model *Y* on the same lane. Figure 13 shows the design of our test case. **Step**-①: *X* and Tesla Model *Y* arrive at the intersection and the traffic light is red. **Step**-②: the traffic light turns green, *X* does not move due to some reasons, thus blocking *Y* from moving forward. Consequently, Tesla Model *Y* remains stopping and waits before the intersection. **Step**-③: after sometime, the light turns RED again, and *X* moves to turn right. **Step**-④: after *X* leaves, Tesla Model *Y* drives into the intersection and runs the RED light.

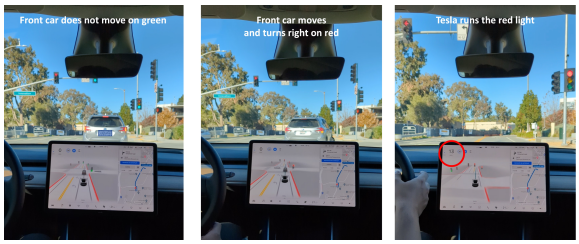


Figure 14: Screenshots of Tesla FSD run-red-light behavior

Fig. 14 presents screenshots of the traffic light signal changes. Tesla Model *Y* ran the red light despite it has enough time and distance to brake and stop. Hence, we have successfully make Tesla Model *Y* violate the traffic rule with the test case derived from our findings in virtual testing. The full video is available in https://drive.google.com/drive/folders/1hqdt2y3HYEkRVru_J8LIy8HUCsZcgoTR.

7 RELATED WORK

Abdessalem et al. [1–3] developed multi-objective methods for searching critical situations and feature interaction failures. Kuutti et al. [23] trained adversarial agents to cause neural network-driven vehicle to collide. Ding et al. [11, 12] created adaptive and multimodal scenario generators to evaluate the search and decision-making algorithms. Koren [22] et al. proposed adaptive stress testing that leverages Monte Carlo Tree search and reinforcement learning to find collision scenarios. O'Kelly et al. [29] established a rare-event simulation framework that tests end-to-end auto-driving vehicle on the highway. Wheeler et al. [36] clustered critical situations regarding frequency and severity, thus accelerating the auto-driving safety testing. Abeysirigoonawardena et al. [4] leveraged Bayesin optimization to generate adversarial scenarios that could increase risk of collisions. AdvSim [35] applied adversarial perturbation to LiDAR data to evaluate collision- and distance-relevant safety potentials. ASF [19] owns a trajectory coverage-guided principle for dynamic scenario parameter fuzzing while AV-Fuzzer [26], *AutoFuzz* [38] and *FusionFuzz* [37] targeted collison-introducing subtle scenarios with evolutionary search. Gambi et al. [16] extracted key factors from crash reports to reconstruct simulation test cases. AsFault [17] leverages procedural content generation and search-based testing to create virtual road networks for challenging the auto-driving lane-keeping ability. Our work differs from existing works that it chews the simulation data from the search of new test cases rather than generating them. Thus, it could complement existing works in a flexible way.

8 CONCLUSION

We propose a method Gatsby for traffic rule safety checking in auto-driving virtual testing. Gatsby eases exisiting ad-hoc testing deficiencies in threefold endeavors. Experiments show that Gatsby can find several new rule violations with tolerable resource cost. Physical test case derived from Gatsby also causes a Tesla FSD vehicle to run a red light, showing the effectiveness of Gatsby.

REFERENCES

[1] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, 63–74.

[2] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 1016–1026.

[3] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 143–154.

[4] Yasasa Abeysirigoonawardena, Florian Shkurti, and Gregory Dudek. 2019. Generating Adversarial Driving Scenarios in High-Fidelity Simulators. In *International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24, 2019*. IEEE, 8271–8277.

[5] Rajeev Alur and David L. Dill. 1994. A Theory of Timed Automata. *Theor. Comput. Sci.* 126, 2 (1994), 183–235.

[6] ApolloAuto. 2021. Apollo Cyber RT. https://cyber-rt.readthedocs.io/en/latest/.

[7] ApolloAuto. 2021. Apollo Dreamland. https://bce.apollo.auto/.

[8] ApolloAuto. 2021. An open autonomous driving platform. https://github.com/ApolloAuto/apollo.

[9] CARLA. 2021. Integration of Autoware AV software with the CARLA simulator. https://github.com/carla-simulator/carla-autoware.

[10] Baiming Chen and Liang Li. 2020. Adversarial Evaluation of Autonomous Vehicles in Lane-Change Scenarios. *CoRR* abs/2004.06531 (2020). arXiv:2004.06531 https://arxiv.org/abs/2004.06531

[11] Wenhao Ding, Baiming Chen, Bo Li, Kim Ji Eun, and Ding Zhao. 2021. Multimodal Safety-Critical Scenarios Generation for Decision-Making Algorithms Evaluation. *IEEE Robotics Autom. Lett.* 6, 2 (2021), 1551–1558.

[12] Wenhao Ding, Baiming Chen, Minjun Xu, and Ding Zhao. 2020. Learning to Collide: An Adaptive Safety-Critical Scenarios Generating Method. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*. IEEE, 2243–2250.

[13] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings (Proceedings of Machine Learning Research, Vol. 78)*. PMLR, 1–16.

[14] 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. *Nature communications* 12 1 (02 2021), 748.

[15] Daniel J. Fremont, Edward Kim, Yash Vardhan Pant, Sanjit A. Seshia, Atul Acharya, Xantha Bruso, Paul Wells, Steve Lemke, Qiang Lu, and Shalin Mehta. 2020. Formal Scenario-Based Testing of Autonomous Vehicles: From Simulation to the Real World. In *23rd IEEE International Conference on Intelligent Transportation Systems, ITSC 2020, Rhodes, Greece, September 20-23, 2020*. IEEE, 1–8.

[16] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 257–267.

[17] Alessio Gambi, Marc Müller, and Gordon Fraser. 2019. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 318–328.

[18] Jiacheng Han and Zhiquan Zhou. 2020. Metamorphic Fuzz Testing of Autonomous Vehicles. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 380–385.

[19] Zhisheng Hu, Shengjian Guo, Zhenyu Zhong, and Kang Li. 2021. Coverage-based Scene Fuzzing for Virtual Autonomous Driving Testing. *CoRR* abs/2106.00873 (2021). arXiv:2106.00873 https://arxiv.org/abs/2106.00873

[20] Zhisheng Hu, Shengjian Guo, Zhenyu Zhong, and Kang Li. 2021. Disclosing the Fragility Problem of Virtual Safety Testing for Autonomous Driving Systems. In *32st IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 24-29, 2021*. IEEE, 59–69.

[21] Jin Hyun Kim, Kim G. Larsen, Brian Nielsen, Marius Mikucionis, and Petur Olsen. 2015. Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools. In *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings (Lecture Notes in Computer Science, Vol. 9128)*. Springer, 47–61.

[22] Mark Koren, Saud Alsaif, Ritchie Lee, and Mykel J. Kochenderfer. 2018. Adaptive Stress Testing for Autonomous Vehicles. In *2018 IEEE Intelligent Vehicles Symposium, IV 2018, Changshu, Suzhou, China, June 26-30, 2018*. IEEE, 1–7.

[23] Sampo Kuutti, Saber Fallah, and Richard Bowden. 2020. Training Adversarial Agents to Exploit Weaknesses in Deep Control Policies. In *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*. IEEE, 108–114.

[24] LGSVL. 2021. Autoware.AI 1.14.0 with LGSVL Simulator. https://www.svlsimulator.com/docs/archive/2020.06/autoware-instructions/.

[25] LGSVL. 2021. Running latest Apollo with LGSVL Simulator. https://www.svlsimulator.com/docs/archive/2020.06/apollo-master-instructions/.

[26] 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 *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*. IEEE, 25–36.

[27] Chengjie Lu, Huihui Zhang, Tao Yue, and Shaukat Ali. 2021. Search-Based Selection and Prioritization of Test Scenarios for Autonomous Driving Systems. In *Search-Based Software Engineering - 13th International Symposium, SSBSE 2021, Bari, Italy, October 11-12, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12914)*. Springer, 41–55.

[28] Microsoft. 2021. Open source simulator for autonomous vehicles built on Unreal Engine / Unity. https://github.com/microsoft/AirSim.

[29] Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, Russ Tedrake, and John C. Duchi. 2018. Scalable End-to-End Autonomous Vehicle Testing via Rare-event Simulation. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 9849–9860.

[30] Pylot. 2021. Modular autonomous driving platform running on the CARLA simulator and real-world vehicles. https://github.com/erdos-project/pylot.

[31] 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 *23rd IEEE International Conference on Intelligent Transportation Systems, ITSC 2020, Rhodes, Greece, September 20-23, 2020*. IEEE, 1–6.

[32] ROS. 2021. Robot Operating System. https://www.ros.org/.

[33] ROS2. 2021. Version 2 of the Robot Operating System (ROS) software stack. https://github.com/ros2.

[34] Udacity. 2021. Udacity: A self-driving car simulator built with Unity. https://github.com/udacity/self-driving-car-sim.

[35] Jingkang Wang, Ava Pun, James Tu, Sivabalan Manivasagam, Abbas Sadat, Sergio Casas, Mengye Ren, and Raquel Urtasun. 2021. AdvSim: Generating Safety-Critical Scenarios for Self-Driving Vehicles. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, 9909–9918.

[36] Tim Allan Wheeler and Mykel J. Kochenderfer. 2019. Critical Factor Graph Situation Clusters for Accelerated Automotive Safety Validation. In *2019 IEEE Intelligent Vehicles Symposium, IV 2019, Paris, France, June 9-12, 2019*. IEEE, 2133–2139.

[37] Ziyuan Zhong, Zhisheng Hu, Shengjian Guo, Xinyang Zhang, Zhenyu Zhong, and Baishakhi Ray. 2021. Detecting Safety Problems of Multi-Sensor Fusion in Autonomous Driving. *CoRR* abs/2109.06404 (2021). arXiv:2109.06404 https://arxiv.org/abs/2109.06404

[38] Ziyuan Zhong, Gail Kaiser, and Baishakhi Ray. 2021. Neural Network Guided Evolutionary Fuzzing for Finding Traffic Violations of Autonomous Vehicles. *CoRR* abs/2109.06126 (2021). arXiv:2109.06126 https://arxiv.org/abs/2109.06126