SALUS: A Novel Data-Driven Monitor that Enables Real-Time Safety in Autonomous Driving Systems

Bohan Zhang, Yafan Huang, Guanpeng Li
Computer Science Department, University of Iowa
Iowa City, IA, USA
{bzhang22, yafan-huang, guanpeng-li}@uiowa.edu

***Abstract*—This paper proposes SALUS, a data-driven real-time safety monitor, that detects and mitigates safety violations of an autonomous vehicle (AV). The key insight is that traffic situations that lead to AV safety violations fall into patterns and can be identified by learning from the safety violations of the AV. Our approach is to use machine learning (ML) techniques to model the traffic behaviors that result in safety violations in the AV, characterize their early symptoms for training a preemptive model, hence deploy and detect real-time safety violations before the actual crashes happen to the AV. In order to train our ML model, we leverage a pipeline of fuzzing techniques to tailor AV-specific safety violation symptoms and generate the training data via data argumentation techniques. Our evaluation demonstrates our proposed technique is effective in reducing over 97.2% of safety violations in industry-level autonomous driving systems, such as Baidu Apollo, with no more than 0.018 false positive values.**

***Index Terms*—Autonomous Vehicles, Safety-Critical Applications, Machine Learning, Software Fuzzing**

I. INTRODUCTION

Autonomous vehicle (AV) technologies, which hold great potential to bring convenience and increase productivity, have entered the public's vision in recent years. Due to the rapid development, there have been many AVs on the market, such as Tesla Model series [1], Waymo Drivers [2], and Baidu Apollo [3]. However, AV industry is still facing many critical challenges – existing AV products are far from being safe on public roads. For example, in a recent accident report from NHTSA, Tesla Autopilot has caused 15 fatal crashes since it was on market in 2015 [4]. Another well-known example is *The death of Elaine Herzberg* [5], which involves a fully self-driving Uber AV in the accident. As a result, the safety of AVs is a major concern to the public, and it largely determines AV's success in the future marketplace.

Traditional methods to improve AV safety are through extensive stress testing (e.g., via road test and simulation-based tests) [6], [2], [7]. After finding the test cases which lead to safety violations, AV developers need to find the code that are responsible for the revealed problems before fixing them in the code. This an iterative process until few safety violations can be found in the AV. While the process has been practiced in the AV industry, it is often impossible to completely eliminate safety violations [8], not to mention it in an agile software development cycle among the rapid evolving AV industry. As a result, the safety of AVs often suffers.

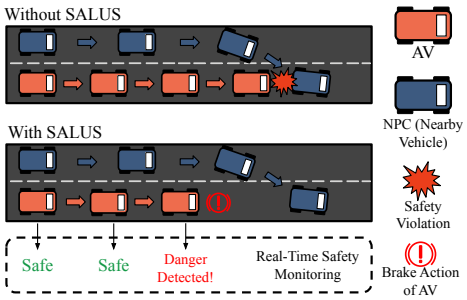


Fig. 1: Detecting and mitigating safety violations with SALUS technique.

In this paper, we propose a safety framework, SALUS, which generates a safety monitor for a given AV. The monitor can detect and mitigate safety violations in an AV in real-time without needing developers to modify AV software. SALUS first generates data which characterizes the safety violations of the AV, then uses ML techniques to craft a model which takes real-time trajectories of surrounding traffic participants to detect the early signs of potential safety violations. In such case, the monitor will alerts the AV, and a mitigation components in SALUS will be triggered and take actions over the AV towards a safe state. Our key insight that lays behind SALUS is that traffic situations which lead to AV safety violations fall into patterns and these patterns can be identified by learning from the safety violations of the AV.

Figure 1 explains the general idea of how the safety monitor generated by SALUS prevents an AV from safety violations in a cut-in scenario (a popular AV safety violation that results in AV at-fault accidents, Section III). The safety monitor evaluates the risks in real-time by analyzing the trajectories of the surrounding vehicles in the first several time slots, and match its pattern with the ML model, in order to determine if any actions of all the participate will likely lead to a safety violation of the AV. If likely dangerous trajectories are found in the surrounding moving vehicles, the monitor preemptively alerts for actions and hence an accident can be avoided. In the example, without SALUS, we find the AV may not be able to tell the danger early enough when the cut-in falls into certain trajectory patterns. Hence, existing AVs usually slow down too late to avoid the crash, hitting the leading vehicle from the back and leading to an at-fault accident.

With SALUS, the safety monitor can detect the early signs of the dangerous trajectory and mitigate (e.g., AV brakes) in a preemptive fashion, in order to avoid the accident.

Our contributions and main results are as follows:

- We conduct an initial study that characterizes safety violations in different AVs. We also stress-test AVs using accident test cases by human drivers based on the NHTSA repository. The initial study shows that different AVs have very different sensitivities to different types of safety violations and human driver accident cases. Therefore, one need to find out safety violations that are specific to the AV under test in order to generate the training data for the ML model that monitors AV safety.
- We design a framework, SALUS, that generates a safety monitor for a given AV. The framework incorporates fuzzing and data argumentation to generate and collect the training data for the safety monitor. Then we craft a sequence-to-sequence model to build our monitor. The entire process is fully automated. The evaluations on three AVs show that our proposed safety monitor is able to detect all the AV safety violations that appear in the stress tests with an only 0.011 false positive on average.
- Upon the alerts sent by the safety monitor, we implement a mitigation strategy which send brake commands to the AV as per situation evolves in order to avoid the potential crash. The evaluation shows that our mitigation strategy, together with the safety monitor, brings AV back to safe state with 99.33% success rate.

The rest of this paper is organized as follows: In Section II, we introduce the background knowledge of this work. In Section III, we discuss about the initial study set up and finding. Section IV we present the detailed design of SALUS. In Section V, we evaluate this work from two perspectives: the prediction accuracy and accident mitigation efficiency. We conclude this work in the last section.

II. BACKGROUND

In this section, we briefly introduce the background knowledge of AV safety, including autonomous driving system, high-fidelity simulator, fuzzing technique, and sequential data processing algorithms.

A. Autonomous Driving Systems

Autonomous driving system (ADS) is a decentralized and highly collaborative combination of all AV-related software and hardware. AVs utilize autonomous driving system (ADS) technology to coordinate with mechanical components and replace human driving [2], [9], [10]. A modern ADS infrastructure consist of a sensor layer and six modules [3], which can be shown in Figure 2 and explained as below.

Sensor Layer contains several sensor units such as IMU, GPS, camera, Radar, and Lidar. These sensors can provide raw data such as pictures, point clouds, GPS locations, etc. All the generated data will be sent to and processed by later modules.

Localization Module obtains the current position information of the vehicle by processing the coordinate system in the high-definition map, GPS information, and the point cloud provided by the Lidar, etc.

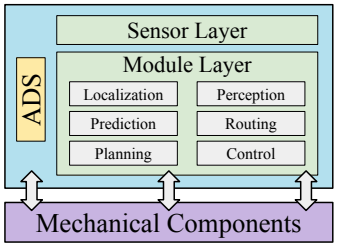


Fig. 2: A high-level overview of ADS.

Perception Module processes the raw data through hardware such as cameras and Lidars carried by the vehicle itself, including data pre-processing (deep learning model) and post-processing. This module can obtain obstacle, lane line, and traffic light information around the vehicle.

Prediction Module obtains the information of obstacles around the vehicle through the perception module and the positioning module and predicts the running trajectory of these obstacles. Those trajectories will be sent to the planning module for calculation for avoiding potential obstacles.

Routing Module calculates the long-term travel route of the vehicle based on current vehicle position, a high-fidelity map, and the destination provided by user.

Planning Module navigates the short-term route, such as avoiding obstacles and following traffic lights, by obtaining the location information and other surrounding environment information provided by other modules.

Control Module is based on the instructions from Planning and Routing modules. It also connects the vehicle hardware, steering wheel, accelerator brake, and other city and county controls of the vehicle through CANbus.

B. LGSVL Simulator

LGSVL (**LG**-**S**ilicon **V**alley **L**ab) [11] is a real-time simulator based on Unity engines and completely simulates our experiments. It can simulate the cars, environments, and traffic participants in real-time. By linking this simulator and ADS through a bridge, LGSVL can return the information of AV driving status, such as speed, position, and steering angle. The AV's action in LGSVL simulator can also be controlled by Python API. In this work, we first build the bridge between LGSVL and Apollo, then run the Python APIs provided by LGSVL to navigate AV. The emergency braking function used in SALUS is also implemented on this API.

C. AV Safety Violations

AV safety violation is the case when AV violates the safety constraints and causes an accident. Safety constraints can be defined from the real-world specific traffic laws and insurance companies' liability judgment. NPC stands for non-player character, in our study, we use NPC to refer to surrounding vehicles. According to the real-world traffic laws, we have two safety constraints in this work. 1) Any car on the road must

Fig. 3: Illustration of LGSVL simulator (left) and Baidu Apollo ADS (right).

keep a safe distance from the front car but fails to do so, likely causing a rear-end collision accident when the front car applies a brake. 2) Any car willing to make a turn on the crossroad or freeway must yield to the straight-going car and rear car but fails to do so, causing a head-on collision and the side collision. Note that we only consider the accident when the AV bears the liability since we could not mitigate the NPC's behavior to avoid the accident when the NPC made a safety violation.

D. Fuzzing for AV Safety Violations

Originated from an operating system academia project [12], software fuzzing has been well studied and widely recognized in test case generation and software vulnerability detection in the past years [13], [14], [15], [16]. By utilizing searching algorithms such as genetic algorithm during software testing phase, researchers can localize vulnerable cases and hence boost the application.

AV-FUZZER [17] is the first work to adopt software fuzzing technique in AV safety and also comprehensively analyzes the efficiency of finding AV-at-fault accidents. AV-FUZZER is based on meta-heuristic algorithms and implements a genetic algorithm to find NPC (nearby vehicles) control instructions that may lead to AV-at-fault accidents. The reasons we utilize AV-FUZZER to generate training data are threefold: (1) Parameters such as initial seed, number of NPCs, and initial locations of AV can be highly-customized in AV-FUZZER. (2) AV-FUZZER is implemented on LGSVL and Baidu Apollo, which are compatible with our testing environments. (3) Compared with random fuzzing technique, AV-FUZZER has a unique fitness function and mutation strategy that can significantly speed up finding the safety violation cases. As a result, we use AV-FUZZER as a black box to generate efficient training data for SALUS with a relatively low cost.

E. Sequence to Sequence Model

Sequence to sequence (Seq2Seq) model [18] is a special class of recurrent neural network (RNN) architectures that are typically utilized to solve complex language problems such as question answering and machine translation. Seq2Seq is composed of an RNN-based encoder and decoder. The encoder maps the sequential information into an encode state, while the decoder transfers this encode state into another vector that contains more refined sequential information. Besides the plain RNN model, other RNN variants such as LSTM [19] and GRU [20] can also be adopted as encoder and decoder components. Since our designed AV safety monitor predicts current surroundings by history information, usually time-series data. We utilize an LSTM-based Seq2Seq model as the main structure of SALUS.

III. INITAL STUDY

In this section, we conduct a comprehensive initial study to investigate the ADSs' sensitivity against different AV safety violations. Knowing the sensitivity is important for us to formulate an efficient strategy when generating training data for our proposed safety monitor.

A. Our Hypothesis

Our hypothesis is that the sensitivity of safety violations are highly ADS-specific. Thus different ADSs, such as different versions, exhibit very different sensitivity to different safety violations. If the hypothesis is true, we need to find AV-specific safety violations given an AV under test. Otherwise, we can use a generic set of training data to train our safety monitor.

B. Experimental Setup

In order to test our hypothesis, we find three versions of ADS, and stress test the ADSs with a set of publicly available test cases provided by Traffic Safety Administration (NHTSA) [4].

1) NHTSA Safety Violation Repository: There are total 37 pre-crash scenarios from a report summarized by NHTSA in April 2007. This report presents comprehensive situations when safety violations are about to occur based on human behaviors. Specifically, the occurrence frequency of these situations includes various nearby environmental factors, such as vehicle position, the status of traffic lights, etc, making our simulations closer to reality. These test cases are also frequently used in other related studies in this area [21], [22].

We select 9 of 37 scenarios as the target violations in our initial study based on the following criteria: (1) We exclude violations caused by the vehicle's problems, such as vehicle puncture or engine failure, since our safety monitors cannot detect the complete functioning of vehicle's components. (2) We do not choose the violations that happen due to environmental issues, such as weather conditions, road smoothness, and whether the road is slippery. (3) We do not select the violations that are collided with non-vehicle objects such as pedestrians, bicycles, street lights, etc. (4) We do not consider the impact of other road signals such as traffic lights and stop signs, etc.

Based on those criteria, we find 9 safety violation types (see Figure 4) from these 37 pre-crash scenarios, which are briefly explained as follow:

- Type 1 **Cut-in**: Two cars are in two lanes in the same direction. AV is in the right lane, while NPC is in the left lane. NPC changes to the right lane in a very short time and decelerates, leaving no time for NPC to make any response. So AV hits the side and rear of NPC.
- Type 2 **Lead Slowdown**: Two cars are in the same lane. AV is in the back, and NPC is in the front. NPC and AV

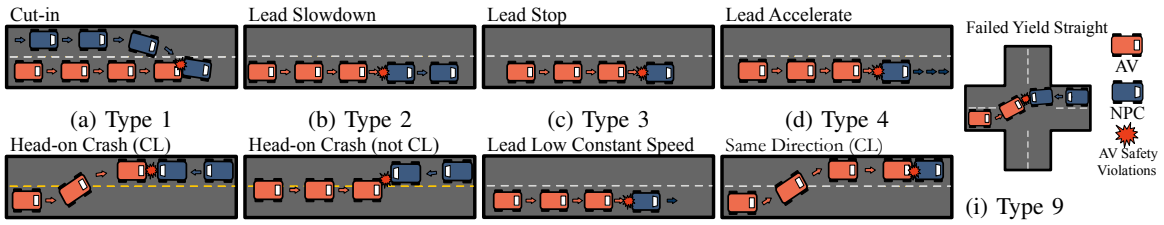


Fig. 4: Illustrations of top 9 safety violation cases selected from NHTSA. "CL" in the figure means "changing lane".

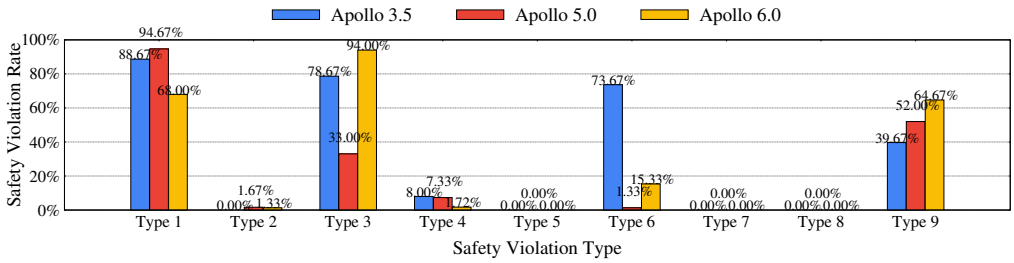


Fig. 5: AV safety violation rates among 9 accident types under three different ADS versions.

are driving at a certain speed, but NPC suddenly brakes to a low speed, leaving no time for AV to react. So AV hits the rear-end of NPC, causing a car accident.

- Type 3 **Lead Stop**: Two cars are in the same lane. AV is in the back, while NPC is in the front. Both NPC and AV are driving at a certain speed, but NPC suddenly brakes to a stop in a very short time, leaving no time for AV to response. So AV collides with NPC, causing a car accident.
- Type 4 **Lead Accelerate**: Two cars are in the same lane. AV is in the back, and NPC is in the front. NPC and AV are both driving at a certain speed, but the NPC suddenly accelerates. AV may accelerate to follow the NPC too much, directly colliding with the NPC vehicle, causing a car accident.
- Type 5 **Head-on Crash (changing lane)**: Two cars are in two lanes in opposite directions. AV tries to switch to another opposite lane and thus head-on crash with NPC.
- Type 6 **Head-on Crash (not changing lane)**: Two cars are in two lanes in opposite directions. AV is too close to NPC in the opposite direction, causing a head-on crash.
- Type 7 **Lead Low Constant Speed**: Two cars are in the same lane. AV is in the back, and NPC is in the front. Both NPC and AV are driving at a certain speed, but NPC is driving at a low and constant speed. The driver of AV may not pay attention to the front car, causing a collision to the rear-end of NPC, incurring a car accident.
- Type 8 **Same Direction (changing lane)**: Two cars are in the same lane. AV is in the back, while NPC is in the front. AV tries to overtake the front NPC but fails to operate the car properly, causing a collision with NPC.
- Type 9 **Failed Yield Straight**: Two cars are in the opposite direction in a non-signalized junction. AV is turning left, while NPC is going straight. AV fails to yield NPC, causing a collision in the side of the NPC.

2) ADS Selection: For the ADS selection, we used ADSs that are from three versions of Baidu Apollo [3], which are 3.5, 5.0, and 6.0, in this section. Baidu Apollo, which is one of the most advanced ADSs in the industry [21], [3], has gone through 7 iterations since its first official release in 2017 yet has implemented a relatively complete functionality. As Baidu Apollo has been commonly-used in this literature, we use its popular branch Apollo 3.5 , Apollo 5.0 and Apollo 6.0 in this work. Each version has different code base.

3) Hardware: Our experiments were conducted on a Linux machine with 32GB RAM. This machine is also equipped with an AMD 5900X CPU (12-core/24-thread) and an NVIDIA GTX 1080 Ti GPU card.

4) Driving Environment Setups: We mock up the NPC trajectory for each accident type in stress test simulations. The time of each simulation trial is 10 seconds. We test 9 accident types in 3 ADS versions, and each accident type has 300 trials on each ADS version. We then compared the safety violation rate in each ADS version. We define the safety violation rate as the number of AV at-fault accidents over the total number of trials in the safety violation type.

C. Results and Observations

We analyze the results from two perspectives: 1) Impact of safety violations on different ADS versions, and 2) Safety violations between human drivers and AVs.

1) Impact on ADS Versions: Figure 5 shows the safety violation rates in different ADS versions caused by different safety violation types. We can observe that the three ADS versions exhibit fairly different sensitivity across each safety violation type. For example, Apollo 6.0 has a 94% safety

violation rate in Type 3 (Lead Stop), whereas they are 78.67% and 33.00% in Apollo 3.5 and Apollo 5.0 respectively. In Type 1 (Cut-in), Apollo 3.5, 5.0 and 6.0 all show high safety violation rates with 88.67%, 94.67%, and 68.00%. To summarize our first observation:

(O-1): Different ADSs shows different vulnerabilities under different safety violation types. Safety monitor should detect specific AV safety violation symptoms for different AVs.

2) Human Drivers and AVs: From the results, we can see that in Type 5 (Head-on Crash), Type 7 (Lead Accelerate), and Type 8 (Lead Low Constant Speed), none of the three ADSes has any safety violations. In another word, those accidents which are proven that human beings are likely to cause, can be avoided by AV itself. Thus, we have our second observation:

(O-2): Human accident insights on avoiding traffic accidents are not entirely suitable for building AV safety monitor. A safety monitor technique for AVs should explore the safety characteristics of AV itself.

In this paper, we propose our technique, SALUS, which is a data-driven ML-based real-time monitor that complies the two observations. Based on (**O-1**), we learn that the safety monitor for a AV need to be based on the set of safety violations that the AV reveal, instead of using a generic set of common safety violations. That leads to a collection of AV-specific data of safety violation given an ADS, in order to train the ML model for safety monitor. On the other hand, for (**O-2**), we know that using existing human driver accidents cannot fully explore the vulnerability of ADS. Further design details in SALUS will be discussed in the next section.

IV. FRAMEWORK DESIGN

We design a framework, SALUS, which can generates a safety monitor for a given ADS, and mitigates AV safety violations in the ADS before they happen. In general, SALUS framework consists of two parts: *Creation* and *Deployment*. The first part presents how an ML-based safety monitor is created, while the second part introduces how to integrate this safety monitor into a running AV, and mitigates AV safety violations. Note that data collection and model training steps can be executed offline, hence they are one-time cost for an ADS.

A. SALUS *Creation*

Figure 6 shows the high-level workflow of the creation process in SALUS. Users only need to provide a specific ADS and SALUS can create a customized ML-based safety monitor for the AV. The creation process can be further divided into two parts: dataset generation and model training. As shown in the figure, dataset generation consists of 1) seed repository, 2) fuzzing engine, 3) data discriminator, and 4) data augmentation to generate the dataset, which are used for training the 5) ML-based model. We present the details of each component.

1) Seed Repository: The seed repository aims to provide the initial seeds for the fuzzing engine. The initial seeds contain the NPCs' behaviors including speeds and turning commands (e.g. line-changing status) at each time slot. These seeds are generated with a random strategy under two user-defined constraints. (1) The speed of NPC is set in the range of 0$\sim$3 km/h, and (2) the turning command is generated as 0, 1, and 2, where 0, 1, and 2 denote "stay in the same lane", "turn left", and "turn right", respectively. Starting for the seeds, the fuzzing engine will start looking for the safety violations that the given ADS has.

2) Fuzzing Engine: The fuzzing engine targets to find AV safety violations by taking the initial seeds as its input. We utilize AV-FUZZER as the main component of fuzzing engine, since it can localize a vast amount of AV safety violations with lower time cost (within only 10 hours) compared with other fuzzing techniques such as random fuzzing. AV-FUZZER is based on meta-heuristic algorithms and adopts a genetic engine to localize accidents by continuously updating the scenario-based fitness scores. As mentioned in Section II-D, we use a state-of-the-art fuzzer, AV-Fuzzer, in this step, in order to find as many and complete safety violations of the AV as possible.

3) Data Discriminator: Data discriminator is designed to group and label AV safety violation cases that are generated by fuzzing engine so that these cases can be used for ML model training. We adopt the K-Means algorithm to cluster the safety violations. Given a set of unlabeled safety violation cases, K-Means algorithm partitions them into several categories by minimizing the within cluster sum of squares (WCSS), and all violations in the same category share the same label, which corresponds to a certain type of safety violations that the AV may reveal.

4) Data Augmentation: The data augmentation component in SALUS is used to increase the size of the training data, thus improves the accuracy of the proposed safety monitor. After the fuzzing engine and data discriminator, we can obtain some labeled AV safety violations in each group. To enrich the data in each group, we perform some random mutations for each of those accidents. These mutations are conducive to maintain the diversity of the training data. The details are explained as follows: For each safety violation group, the NPC speed is randomly updated by adding or subtracting 10%. Those generated potential safety violations in the group will be tested with the ADS under test again and so to confirm the label. If the new data indeed leads to a safety violation, we add it to the group. Otherwise, the case will be discarded.

5) ML-Based Model: ML-based model is the key component of the safety monitor. We adopt the long short-term memory (LSTM) sequence to sequence (Seq2Seq) model along with two fully-connected layers (FCLs) for the model. The monitor predicts a safety factor in each time slot (1 time slot = 0.25 seconds in our experiment). The safety factor is a metric we use in the ML output to gauge how likely the current trajectory symptoms lead to a safety violation of the AV. We first pre-process the data we generate before starting training the model. By doing so, the raw data can be then transferred into well-constructed feature vectors that can be easily trained and inferenced by our ML model. Take k-th time slot in j-th trial for example, the processed data can be represented as $I^{jk} = [i_0^{jk}, i_1^{jk}, ..., i_4^{jk}]$. Among those factors,

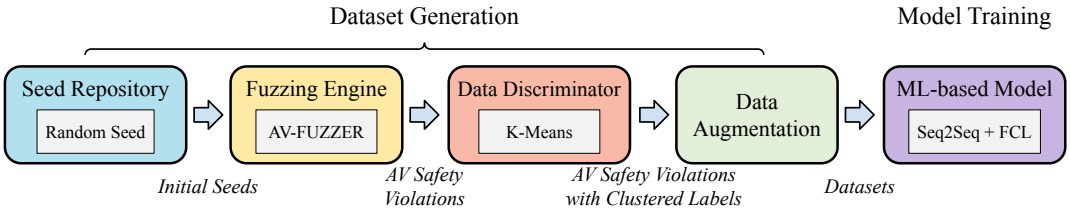


Fig. 6: The workflow of SALUS model creation phase

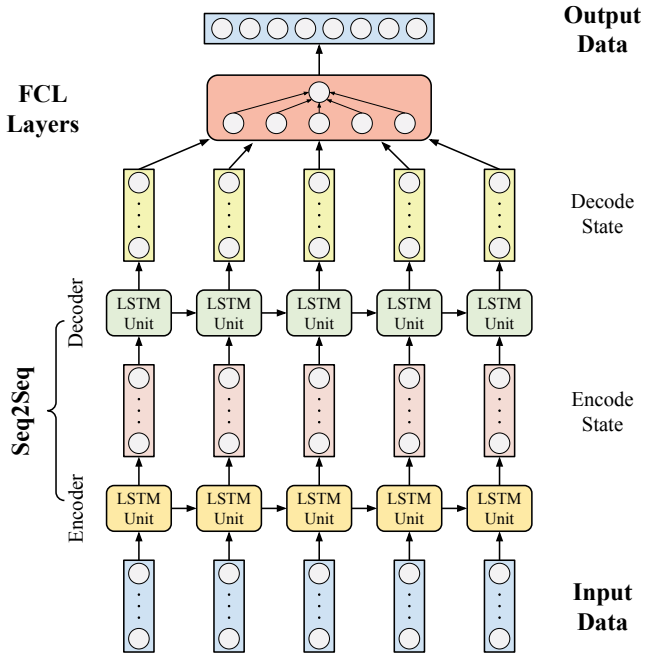


Fig. 7: The network structure of ML-based model. "Seq2Seq" and "FCL Layers" in the left represent the model flow, while "Input/Output data" and "Encode/Decode state" in the right explain the data flow.

i_0^{jk} denotes the Euclidean distance between the position of the AV and the NPC; i_1^{jk} and i_2^{jk} represent the running speed of these two vehicles; i_3^{jk} denotes the relative steering angle of the AV and the NPC; and i_4^{jk} is a coefficient that records the relative position information of these two vehicles.

Figure 7 presents the architecture of our ML model. The model takes information from 5 consecutive time slots (4 previous and 1 current slots) as the input, which compose a 5×5 matrix $\mathbb{I}^{jk} = [I^{jk-4}, I^{jk-3}, ..., I^{jk}]^{\mathrm{T}}$. And the output $\mathbb{O}^{jk}$, which is a vector with a length of 8, is the predicted safety factors of following 8 time slots. As Figure 7 shows, Seq2Seq in the ML model consists of two important layers: an encoder and a decoder. The encoder calculates the input data into a hidden encode state, while the decoder takes hidden encode state as input and transfers it to the decode state. Each computation unit in the encoder and decoder is based on LSTM. Still using I^{jk} to explain, the computation process of one LSTM unit can be formulated as below:

$$
\begin{aligned}
f_q &= \sigma(W_f \cdot (E^{jk-1} \oplus I^{jk}) + b_f) \\
i_q &= \sigma(W_i \cdot (E^{jk-1} \oplus I^{jk}) + b_i) \\
\widetilde{C}_q &= \tanh(W_C \cdot (E^{jk-1} \oplus I^{jk}) + b_C) \\
C_q &= C_{q-1} \odot f_q + \widetilde{C}_q \odot i_q \\
o_q &= \sigma(W_o \cdot (E^{jk-1} \oplus I^{jk}) + b_o) \\
E^{jk} &= o_q \odot \tanh(C_q)
\end{aligned}
\tag{1}
$$

where E^{jk} and E^{jk-1} are the output vectors of current and the previous LSTM units with length 5; f_q, i_q, and o_q denote forget, input, and output gates; $\widetilde{C}_q$ and C_q denote represent candidate values vector and memory state vector; W_f, W_i, W_C and W_o are the weight matrices initialized with random values, while b_f, b_i, b_C and b_o are bias vectors of each gate or cell; σ and tanh denote sigmoid and hyperbolic tangent activation function; $\odot$ and $\oplus$ represent hadamard product and concatenation respectively. Thus, the LSTM-based encoder can compute input matrix $\mathbb{I}^{jk}$ into $\mathbb{E}^{jk}$, of which dimension is also 5×5. For simplicity, we formulate this process as $\mathbb{E}^{ji} = \mathrm{LSTM}(\mathbb{I}^{jk})$. Then, an LSTM-based decoder decodes the encode state matrix $\mathbb{E}^{jk}$ into the decode state matrix, of which dimension is 5×8. At last, FCL layers map the decode state matrix into model output $\mathbb{O}^{jk}$, which contains the predicted safety factors of AV for the following 8 time slots (from $k+1$ to $k+8$). In all, the ML model can be represented by the following equation:

$$
\mathbb{O}^{jk} = W^{\mathrm{T}} \mathrm{LSTM}_2(\mathrm{LSTM}_1(\mathbb{I}^{jk})) \tag{2}
$$

where LSTM_1 and LSTM_2 denote the encoder and decoder of Seq2Seq, and W represent the coefficient weights of FCL layers with dimension 5×1.

B. Safety Monitor Deployment and Safety Violation Mitigation

Figure 9 presents how we deploy our safety monitor generated by SALUS on an AV to detect and mitigate safety violations in real-time. When AV starts cruising, SALUS first collects 1.25 seconds of data as the initial input to the safety monitor – this short period of time refer to the cold start phase in the monitor. Such a cold start cost is very small and can be neglected in terms of the safety perspective. Then, the monitor will collect the real-time information of surrounding NPC trajectories and perform inference through the ML model every time slot (0.25 seconds). Once the predicted safety factor is higher than the user-defined threshold, the monitor will alert. Note that each ML-based safety monitor holds its own

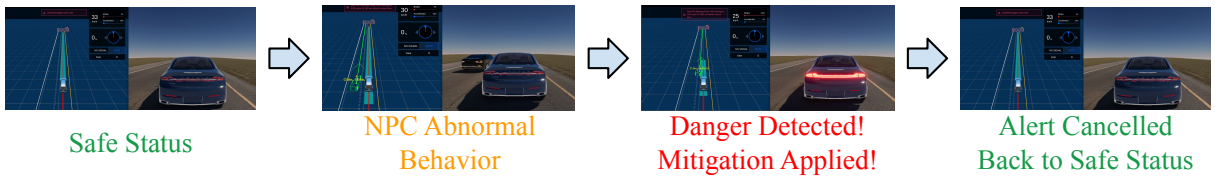


Fig. 8: A demo that shows how SALUS cooperates with Baidu Apollo.

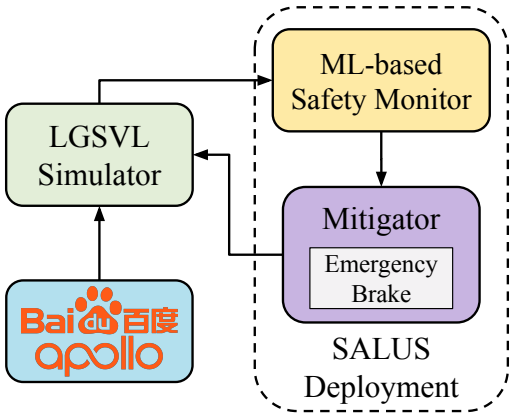


Fig. 9: Deployment of the safety monitor and mitigator by SALUS

threshold in practice, and we carefully explain and evaluate the value selections of the threshold in Section V-C. For the mitigation, as shown in Figure 8, the monitor will take over the control of the AV and apply a brake commands until our safety monitor shows that the trajectories resume back to a safety state.

V. EVALUATION

In this section, we first present the driving environment setups and the evaluation metrics. Then we evaluate SALUS based on two metrics: (1) the prediction accuracy of safety violations, and (2) the success rate of safety violation mitigation. Besides, we also conduct cross validation with different models and ADS versions.

A. Driving Environment Setup

The ML-based safety monitor is developed under PyTorch 1.11. For all 3 Baidu Apollo versions, we conduct the model creation and model deployment under the same initial seed. For the map selection, we use a standard two-lane road structure – similar roach structures are commonly used in related works as well [17], [10]. We run the fuzzing engine for 10 hours and find 9∼14 safety violations for each ADS under test. These safety violations fall into 2∼4 groups by data discriminator. In detail, Apollo 3.5, 5.0, and 6.0 have 2 groups, 3 groups, and 4 groups, respectively.

B. Evaluation Metrics

We evaluate SALUS using the following metrics:

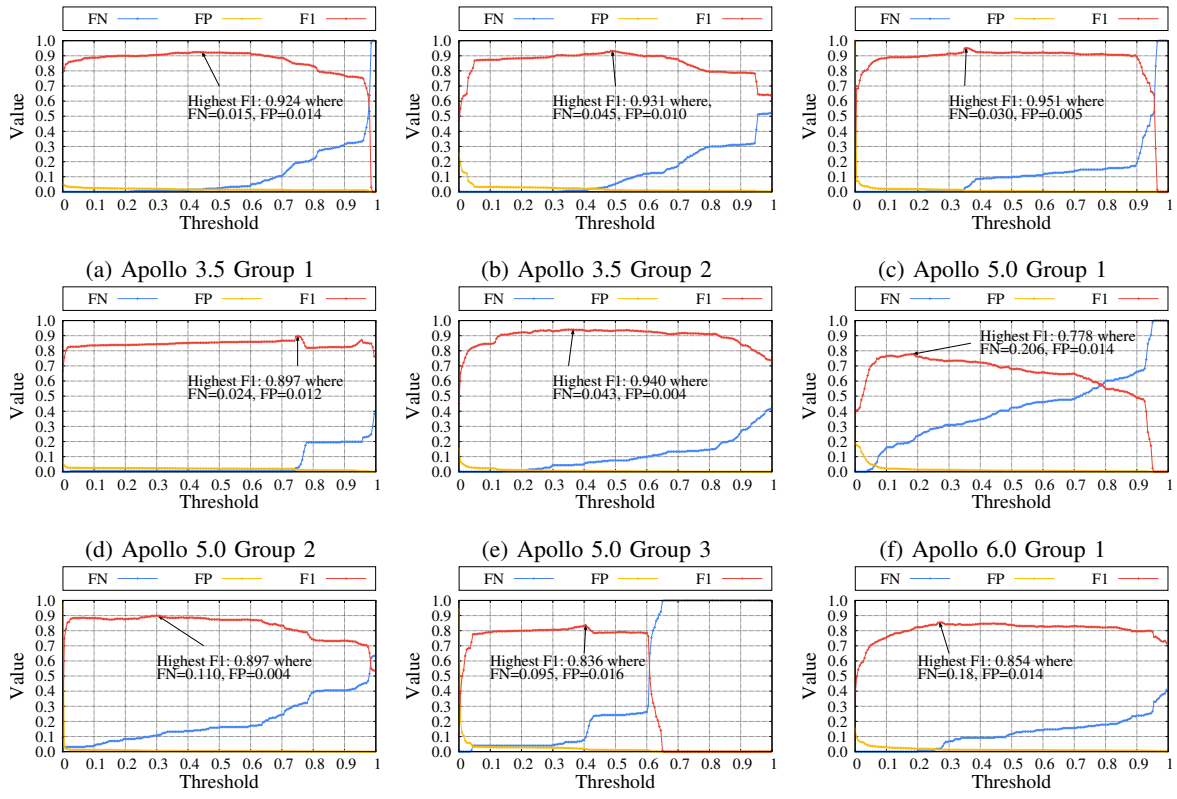
- **False Positive (FP)**: FP can be calculated by (number of FP time slots) / (number of total time slots). It quantifies how much SALUS falsely predict safe scenarios as dangerous. The lower the better.
- **False negative (FN)**: FN can be calculated by (number of FN trials) / (number of total trials). It quantifies how much SALUS falsely predict dangerous scenarios. The lower the better.
- **F1 Score (F1)**: F1 can be calculated as formulated as below equation. It is the primary metrics to evaluate the performance of a classifier [23]. We also use F1 score to determine the threshold in our ML-based safety monitor.

$$\mathrm{F1} = \frac{\mathrm{TP}}{\mathrm{TP} + \frac{1}{2}(\mathrm{FP} + \mathrm{FN})} \tag{3}$$

  where TP indicates true positive (i.e. correctly predict safe as safe). The higher (closer to 1) the better.
- **Safety Violation Rate**: Fault rate can be calculated by (number of safty violations trials) / (number of total trials). It is the most intuitive metrics to evaluate the safety indicator of an AV (the lower the better). The lower safety violation rate of an AV is, the more safety violation that AV can avoid. The lower (closer to 0) the better.

C. Prediction Accuracy

Before we present the prediction accuracy, we first present the detailed settings. Back to the data augmentation phase in SALUS creation workflow, we collected nearly one thousand trials of AV safety violations and other actions for each group. If one trial is very close to a safety violation but finally executes in normal, we define it as a close trial. On the one hand, we use 400 AV safety violation trials, 200 close trials, and 200 random trials for training the ML predictor for each group. On the other hand, we use another 100 AV safety violation trials, 50 close trials, and 50 random trials for testing. We balance the training and testing data to obtain a higher prediction accuracy and a fair evaluation, respectively. We decompose each trial into several sliding windows. There are 40 time slots in each trial. The first 5 time slots are the model input (i.e. cold-start stage) for SALUS, while the output is the safety prediction for the later 8 time slots. Recall that the output of the ML-based model is an 8 dimension vector, where each number in this vector represents a safety prediction for its corresponding time slot. Specifically, this predicted



(a) Apollo 3.5 Group 1 (b) Apollo 3.5 Group 2 (c) Apollo 5.0 Group 1

(d) Apollo 5.0 Group 2 (e) Apollo 5.0 Group 3 (f) Apollo 6.0 Group 1

(g) Apollo 6.0 Group 2 (h) Apollo 6.0 Group 3 (i) Apollo 6.0 Group 4

Fig. 10: Prediction accuracy of each group across different thresholds in each ADS version.

number is a floating-point number in range of 0 to 1. We set a hyperparameter **threshold** for each group here. If this number exceeds the user-defined threshold, the corresponding time slot will be regarded as dangerous, otherwise safe. **False positive**: If any of the 8 predicted numbers exceeds the threshold and the ground truths of them are all 0, such will be recorded as a false positive case. **False negative**: If all of the 8 predicted numbers do not exceed the threshold while there is one number or more has a ground truth 1, such will be recorded as a false negative case. For each group, we traverse the thresholds from 0 to 1 and choose a value that can lead to the largest F1 score, which is the prime metrics to evaluate the effectiveness of a classifier. Such an optimized value will be selected in SALUS deployment. If one ML model of any group evaluates NPC actions in the current surrounding as dangerous, the message will be sent to mitigator. And SALUS will alert ADS and then conduct an emergent brake.

Figure 10 presents the comprehensive prediction results. Since we adopt three Apollo versions and each version has different number of groups, there are 9 ML-based model presented. As we see, F1 in ML-based models keep in a high value ($\geq$ 0.7) in most of the threshold, demonstrating the effectiveness of the ML-based model design. We also observe that in some groups, such Apollo 6.0 Group 3, the F1 score decrease rapidly when we increase the threshold from 0.5 to 0.6. The reason is that the data points gathered at 0.5$\sim$0.6 in this group, resulting in a rapid change in this range even we switch the threshold very slightly. However such rapid decreases does not affect the functionality of the safety monitor, since this monitor is only deployed with the threshold that has highest F1. Table I presents the F1 values that are used for model deployment for each group. Note that we do not calculate the average threshold as it does not hold practical meanings. As we can see, the F1 has an average value of 0.889 and is in the range of 0.778 to 0.951 across 9 ML-based models, which demonstrates that the prediction of our proposed ML-based safety monitor is accurate.

TABLE I: F1 and its corresponding threshold in safety monitor deployment.

Model Name	Threshold	F1
Apollo 3.5 Group 1	0.445	0.924
Apollo 3.5 Group 2	0.490	0.931
Apollo 5.0 Group 1	0.355	0.951
Apollo 5.0 Group 2	0.750	0.897
Apollo 5.0 Group 3	0.365	0.940
Apollo 6.0 Group 1	0.185	0.778
Apollo 6.0 Group 2	0.345	0.897
Apollo 6.0 Group 3	0.405	0.836
Apollo 6.0 Group 4	0.270	0.854
Average	-	**0.889**

Results: The proposed ML-based safety monitor in SALUS is accurate in predicting AV safety violations with an average of 0.889 F1 value across all ADS versions.

D. Safety Violation Mitigation

We evaluate the safety violation mitigation efficiency from two perspectives: safety violation rate (how many safety violations can be mitigated) and FP value (the safe scenario will not be predicted as dangerous).

1) Safety Violation Rate: We perform ablation experiments to examine the violation mitigation efficiency of SALUS. Specifically, we evaluate safety violation rates with and without SALUS protection in dangerous scenarios for different ADS versions. To generate enough dangerous scenarios for evaluation, we follow the same strategy in data augmentation (see Section IV-A4). For each group, we generate 200 trials that will lead to safety violations. Since Apollo 3.5, 5.0, and 6.0 have 2, 3, and 4 groups, there will be 400, 600, and 800 trials for each ADS version, respectively. All generated trials will be evaluated twice: one for ADS without SALUS protection and the other for ADS with SALUS deployed.

Figure 11 presents the results. In all, we can observe that our SALUS is able to mitigate at least 97.2% of safety violation cases in all ADS versions compared to the ADS without any protections. In Apollo 3.5, the safety violation rate without SALUS is 62.50% (250/400), while this number decreases to 1.75% (7/400) after we deploy SALUS protection, mitigating 97.2% safety violations. In Apollo 5.0 and 6.0, the safety violation rates reduce from 50.60% (304/600) to 0.00% (0/600) and 27.63% (221/800) to 0.25% (2/800), successfully mitigating 100% and 99.10% safety violations, respectively.

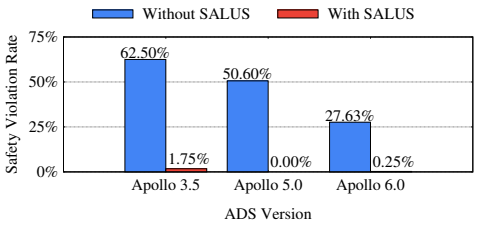


Fig. 11: Safety violation rates across different ADS Versions.

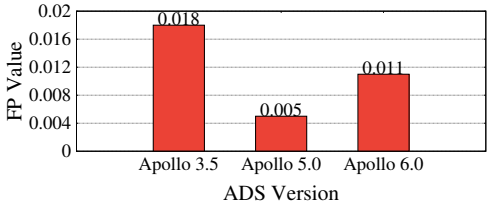


Fig. 12: FP value with SALUS across different ADS versions.

2) False Positive (FP): We also evaluate the FP value of SALUS technique to demonstrate that SALUS not only can mitigate safety violations but also reacts normally in those safe scenarios. Before we present the results, we first introduce how we create testing data of safe scenarios. We modify the convergence condition of fuzzing engine (see Section IV-A2) and randomly generate 200 trials, which contain the speed and other behaviors of NPC. These trials are also executed in the LGSVL simulator under three ADS versions to make sure they will not result in any safety violations. We maintain 35 time slots for each trial. Therefore, there are 7000 (200×35) time slots in total for evaluating the FP across three versions of ADS.

Figure 12 presents the results. Among 7000 time slots to be tested, Apollo 3.5, 5.0, and 6.0 only report 124, 37, and 79 times of false alerts, which corresponds to 0.018, 0.005, and 0.011 FP values. These FP values are small enough and routinely regarded as a convincing classifier [24]. In all, we observe that all three versions of ADS can keep a low FP value of no more than 0.018.

Results: SALUS can mitigate more than 97.2% of AV safety violations compared with an ADS without any protection, and the FP value is no more than 0.018.

E. Cross-Validation of ML-based Models and ADS Versions

We perform cross-validation tests across each ML-based model and each ADS version, to demonstrate each ML-based model is more suitable for its corresponding ADS version. Similar to previous evaluations, we perform cross-validation based on two important metrics, safety violation rates and FP values. The only difference is each ML-based model is also deployed to all three selected ADS versions. Tables II and III present the cross-validation results. In Table II, we can see that, in Apollo 5.0 and 6.0, the lowest safety violation rates are found in its corresponding ADS version. The same observation can be obtained in Table III as well, where the lowest FP values for Apollo 3.5, 5.0, and 6.0 are 0.018, 0.005, and 0.011 respectively. These FP values also lie on their corresponding ADS versions. The only exception can be found in the safety violation rate of Apollo 3.5, where the models for Apollo 5.0 and 6.0 can also achieve low safety violation rates (0.00% and 0.75%). The key reason is that Apollo 5.0 and 6.0 preserve the driving features of Apollo 3.5, which allow the characteristics learned from models for these two versions to be compatible with Apollo 3.5. However, we argue that such exceptions do not affect the main results, since the model for Apollo 3.5 has the lowest FP value on its corresponding ADS version and similar observations cannot be obtained through the upgrades from 5.0 to 6.0.

Results: The ML-based model always achieves its best performance only on its corresponding ADS version.

VI. RELATED WORK

Existing works that enable AV safety are mostly based on the testing-debugging strategy. After the safety violations are found through stress testings, AV developers need to localize

TABLE II: The cross validation of safety violation rates across different models and ADS versions.

	Apollo 3.5	Apollo 5.0	Apollo 6.0
Model for Apollo 3.5	**1.75%**	6.24%	7.18%
Model for Apollo 5.0	0.00%	**0.00%**	5.55%
Model for Apollo 6.0	0.75%	0.60%	**0.25%**

TABLE III: The cross validation of FP values across different models and ADS versions.

	Apollo 3.5	Apollo 5.0	Apollo 6.0
Model for Apollo 3.5	**0.018**	0.069	0.072
Model for Apollo 5.0	0.027	**0.005**	0.056
Model for Apollo 6.0	0.086	0.060	**0.011**

the corresponding code segment and then fix the bugs. Fremont et al [6] introduce an ML-based probabilistic programming language, Scenic, which generates testing scenarios that can help debug the perception module in ADS. Garcia et al [8] analyze the existing data reports of many AV accidents, locate the modules with errors directly, and summarize 16 findings through comprehensive evaluations, which can bring insights for subsequent bug fixing. Horel et al [25] propose a decision-making system for AVs to enable AV safety based on Monte Carlo sampling and deep reinforcement learning searching strategies. Tang et al [26] conduct a simulation-based test based on a behavior tree to generate all possible violations that AV could carry out. Ghodsi et al [27] enable AV safety via analyzing real-world driving data in an industrial-level high-definition simulator. Although these methods achieve promising performance in generating testing cases for enabling AV safety, these methods still require massive human-supervised code localizing and bug fixing efforts. In contrast, our proposed protection technique SALUS does not require any bug fixing stages and can be integrated with ADS flexibly, significantly reducing the cost.

VII. Conclusion and Future Work

In this work, we propose SALUS technique enable AV safety by detecting and mitigating safety violations in real-time. SALUS uses an ML-based safety monitor composed of an LSTM-based Seq2Seq and several FCL layers to evaluate the risks of surroundings. We present the creation and deployment stages of SALUS technique. Both of these two stages are automatic and flexible, significantly reducing the time cost compared with existing AV protection methods. Evaluation shows that the ML model is accurate in predicting safety violations and SALUS is effective in mitigating those violations (more than 97.2% accident cases) in real-time driving scenarios for different versions of Baidu Apollo.

In the future, we plan to conduct our research based on two directions. (1) Accelerating the fuzzing engine in SALUS creating workflow. In this work, we only choose AV-FUZZER for generating training the dataset for SALUS, since it can localize AV safety violations with a low time cost. However, this cost can be further improved by adopting more fine-grained mutation techniques. (2) Enriching the mitigation strategy. In this work, we only adopt the emergent brake in mitigating safety violations. There are also some other means, such as different levels of braking and direction changing, for enabling AV safety under various scenarios. We will include these methods in the future.

References

[1] "Tesla death," https://www.tesladeaths.com.

[2] A. C. Madriga, "Inside waymo's secret world for training self-driving cars," *The Atlantic*, vol. 23, 2017.

[3] "Baidu apollo github repository," https://github.com/ApolloAuto/apollo.

[4] "Pre-crash scenario typology for crash avoidance research," https://rosap.ntl.bts.gov/view/dot/6281.

[5] "Uber's self-driving operator charged over fatal crash," https://www.bbc.com/news/technology-54175359.

[6] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: a language for scenario specification and scene generation," in *PLDI 2019*.

[7] R. Lee, O. J. Mengshoel, A. Saksena, R. Gardner, D. Genin, J. Silbermann, M. Owen, and M. J. Kochenderfer, "Adaptive stress testing: Finding failure events with reinforcement learning," *arXiv 2018*.

[8] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, Chen, and Q. Alfred, "A comprehensive study of autonomous vehicle bugs," in *ICSE 2020*.

[9] F. Zhu, L. Ma, X. Xu, D. Guo, X. Cui, and Q. Kong, "Baidu apollo auto-calibration system-an industry-level data-driven and learning based vehicle longitude dynamic calibrating algorithm," *arXiv 2018*.

[10] S. Jha, S. Banerjee, T. Tsai, S. K. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Ml-based fault injection for autonomous vehicles: A case for bayesian fault injection," in *DSN 2019*.

[11] "Lgsvl simulator," https://www.lgsvlsimulator.com/.

[12] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM 1990*.

[13] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "{MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *USENIX Security 2020*.

[14] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in *S&P 2020*.

[15] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 533–544.

[16] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "{SpecFuzz}: Bringing spectre-type vulnerabilities to the surface," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1481–1498.

[17] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, "Av-fuzzer: Finding safety violations in autonomous driving systems," in *ISSRE 2020*.

[18] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *NIPS 2014*.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation 1997*.

[20] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv 2014*.

[21] "Carla autonomous driving system," https://carla.readthedocs.io//.

[22] R. R. Lutz, "Safe-ar: Reducing risk while augmenting reality," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 70–75.

[23] A. Tharwat, "Classification assessment methods," *Applied Computing and Informatics*, 2020.

[24] A. Fisch, T. Schuster, T. Jaakkola, and R. Barzilay, "Conformal prediction sets with limited false positives," *arXiv 2022*.

[25] J.-B. Horel, C. Laugier, L. Marsso, R. Mateescu, L. Muller, A. Paigwar, A. Renzaglia, and W. Serwe, "Using formal conformance testing to generate scenarios for autonomous vehicles," in *DATE 2022*.

[26] Y. Tang, Y. Zhou, F. Wu, Y. Liu, J. Sun, W. Huang, and G. Wang, "Route coverage testing for autonomous vehicles via map modeling," in *ICRA 2021*.

[27] Z. Ghodsi, S. K. S. Hari, I. Frosio, T. Tsai, A. Troccoli, S. W. Keckler, S. Garg, and A. Anandkumar, "Generating and characterizing scenarios for safety testing of autonomous vehicles," in *IV 2021*.