# ML-based Fault Injection for Autonomous Vehicles:
## A Case for Bayesian Fault Injection

Saurabh Jha*, Subho S. Banerjee*, Timothy Tsai†, Siva K. S. Hari†, Michael B. Sullivan†,
Zbigniew T. Kalbarczyk*, Stephen W. Keckler† and Ravishankar K. Iyer*
*University of Illinois at Urbana-Champaign, Urbana-Champaign, IL 61801, USA.
†NVIDIA Corporation, Santa Clara, CA 94086, USA.

*Abstract*—**The safety and resilience of fully autonomous vehicles (AVs) are of significant concern, as exemplified by several headline-making accidents. While AV development today involves verification, validation, and testing, end-to-end assessment of AV systems under accidental faults in realistic driving scenarios has been largely unexplored. This paper presents DriveFI, a machine learning-based fault injection engine, which can mine situations and faults that maximally impact AV safety, as demonstrated on two industry-grade AV technology stacks (from NVIDIA and Baidu). For example, DriveFI found 561 safety-critical faults in less than 4 hours. In comparison, random injection experiments executed over several weeks could not find any safety-critical faults.**

*Index Terms*—**Autonomous Vehicles; Fault Injection**

## I. Introduction

Autonomous vehicles (AVs) are complex systems that use artificial intelligence (AI) and machine learning (ML) to integrate mechanical, electronic, and computing technologies to make real-time driving decisions. AI enables AVs to navigate through complex environments while maintaining a *safety envelope* [1], [2] that is continuously measured and quantified by onboard sensors (e.g., camera, LiDAR, RADAR) [3]–[5]. Clearly, the safety and resilience of AVs are of significant concern, as exemplified by several headline-making AV crashes [6], [7], as well as prior work characterizing AV resilience during road tests [8]. Hence there is a compelling need for a comprehensive assessment of AV technology.

AV development today involves verification [9]–[12], validation [13], and testing [14], [15] as well as other forms of assessment throughout the life cycle. However, assessment of these systems in realistic execution environments, especially because of the occurrence of random faults, has been challenging. Fault injection (FI) is a well-established method for testing the resilience and error-handling capabilities of computing and cyber-physical systems [16] under faults. FI-based assessment of AVs presents a unique challenge not only because of AV's complexity but also because of the centrality of AI in a free-flowing operational environment [17]. Also, AVs represent a complex integration of software [18] and hardware technologies [19] that have been shown to be vulnerable to hardware and software errors (e.g., SEUs [20], [21], *Heisenbugs* [22]). Future trends of increasing code complexity and shrinking feature sizes will only exacerbate the problem.

This paper presents *DriveFI*, an intelligent FI framework for AVs that addresses the above challenge by identifying hazardous situations that can lead to collisions and accidents. DriveFI includes (a) an FI engine that can modify the software and hardware states of an autonomous driving system (ADS) to simulate the occurrence of faults, and (b) an ML-based fault selection engine, which we call *Bayesian fault injection*, that can find the situations and faults that are most likely to lead to violations of safety conditions. In contrast, traditional FI techniques [16] often do not focus on safety violations, and in practice have low manifestation rates and require enormous amounts of time under test [23], [24]. Note that given a fault model, DriveFI can also perform random FI to obtain a baseline.

**Contributions.** DriveFI's Bayesian FI framework is able to find safety-critical situations and faults through causal and counter-factual reasoning about the behavior of the ADS under a fault. It does so by (a) *integrating domain knowledge* in the form of vehicle kinematics and AV architecture, (b) *modeling safety* based on lateral and longitudinal stopping distance, and (c) using *realistic fault models* to mimic soft errors and software errors. Items (a), (b), and (c) are integrated into a *Bayesian network* (BN). BNs provide a favorable formalism in which to model the propagation of faults across AV system components with an interpretable model. The model, together with fault injection results, can be used to design and assess the safety of AVs. Further, BNs enable rapid probabilistic inference, which allows DriveFI to quickly find safety-critical faults. The Bayesian FI framework can be extended to other safety-critical systems (e.g., surgical robots). The framework requires specification of the safety constraints and the system software architecture to model causal relationship between the system sub-components. We demonstrate the capabilities and generality of this approach on two industry-grade, level-4 ADSs [25]: DriveAV [3] (a proprietary ADS from NVIDIA) and Apollo 3.0 [4] (an open-source ADS from Baidu).

**Results.** We use three fault models: (a) random and uniform faults in non-ECC-protected processor structures, (b) random and uniform faults in ADS software module outputs (corrupted with min or max values), and (c) faults in which ADS module outputs are corrupted with Bayesian FI. The major results of our injection campaigns include:

- Using fault model (b) we compiled a list of 98,400 faults. An exhaustive evaluation of all 98,400 faults in our simulated driving scenarios would have taken 615 days. In comparison, our Bayesian FI was able to find 561 faults that maximally impact AV safety in less than 4 hours. Thus, Bayesian FI achieves 3690× acceleration. Two cases found by Bayesian FI are described in §II-D; one, in particular, mimics the Tesla vehicle crash [6].
- Bayesian FI is able to find critical faults and scenes that led to safety hazards. (a) Out of the 561 identified faults, 460 manifested as safety hazards. (b) These 460 faults were found to be associated with 68 safety-critical scenes[1] (out

[1] A scene is represented by one camera frame.

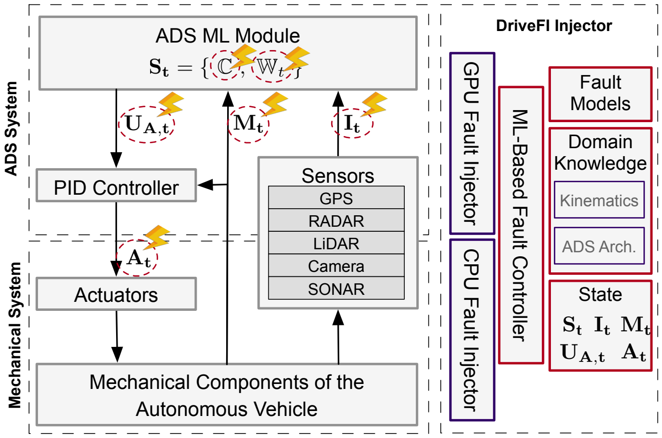


**Figure 1:** A high-level overview of the AV's autonomous and mechanical systems, and its interaction with DriveFI.

of 7200 scenes).

- In comparison, several weeks of 5000 random FI experiments did not result in discovery of a single safety hazard. Only 1.93% of the single-bit injections led to silent-data corruption (SDC) that caused actuation errors. The ADS recovered from all of these errors without any safety violations. In 7.35% of the FIs, kernel panics and hangs occurred. It is expected that recovery from such faults can be done with the backup/redundant systems that are present in AVs today.

We believe that the mining of critical situations by Bayesian FI will have wider applicability beyond our fault injections here. Combining results from a range of fault injection experiments to create a library of situations will help manufacturers to develop rules and conditions for AV testing and safe driving.

**Putting DriveFI in Perspective.** Early work studied the safety of AVs using system-theoretic approaches [26], [27]. More recent studies have focused on the resilience of constituent modules of an ADS (described in §IV), e.g., [24], [28]–[30]. Another line of work [31], [32] has used FI to study sensor-related resilience in AVs. In contrast to DriveFI, none of the prior approaches have considered the resilience of modern end-to-end AI-driven systems that use industry-grade ADSs to mine faults that lead to safety hazards.

## II. Approach Overview

This section provides an overview of the AI-driven Bayesian FI approach advocated in this paper. We now introduce the formalism that is used in the remainder of the paper.

### A. Autonomous Driving System

Fig. 1 illustrates the basic control architecture of an AV (henceforth also referred to as *Ego Vehicle*, EV). It consists of mechanical components and actuators that are controlled by an *ADS*, which represents the computational (hardware and software) component of the AV. At every instant in time, $t$, the ADS system takes input from sensors $\mathbf{I_t}$ (e.g., cameras, LiDAR, GPS), takes inertial measurements $\mathbf{M_t}$ from the mechanical components (e.g., velocity $v_t$, acceleration $a_t$), and infers actuation commands $\mathbf{A_t}$ (e.g., throttle $\zeta$, brake $b$, steering angle $\phi$). For clarity, we further subdivide the ADS into two components: (a) an ML module (responsible for perception and planning) that takes as inputs $\mathbf{I_t}$ and $\mathbf{M_t}$ and produces raw-actuation commands $\mathbf{U_{A,t}}$, and (b) a PID controller [33] that is responsible for smoothing the output $\mathbf{U_{A,t}}$ to produce $\mathbf{A_t}$. The PID controller ensures that the AV does not make any sudden changes in $\mathbf{A_t}$. The ADS ML module has an instantaneous state $\mathbf{S_t}$ that consists of configuration parameters $\mathbb{C}$ (e.g., neural network weights to perceive input camera data) and a *world model* $\mathbb{W}_t$, which maintains and tracks the trajectories of all static objects (e.g., lane markings) and dynamic objects (e.g., other vehicles) perceived by the ADS.

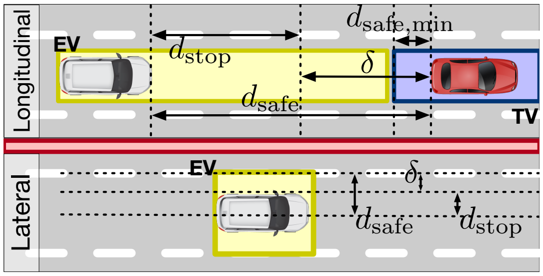


**Figure 2:** Definition of $d_{\text{stop}}$, $d_{\text{safe}}$, and $\delta$ for lateral and longitudinal movement of the car. Non-AV vehicles are labeled as *target vehicles* (TV).

### B. Safety

We define the instantaneous safety criteria of an AV in terms of the longitudinal (i.e., direction of motion of the vehicle) and lateral (i.e., perpendicular to the direction of the vehicle motion) Cartesian-distance travelled by the AV (see Fig. 2). Those criteria form a "primal" definition of safety based on collision avoidance, which can be extended with other notions of safety, e.g., using traffic rules. The extended notions of safety are not considered in this paper, as they can be nuanced based on the laws of the geographic regions in which they are applied.

**Definition 1.** The *stopping distance* $d_{\text{stop}}$ is defined as the maximum distance the vehicle will travel before coming to a complete stop while the maximum comfortable deceleration $a_{\text{max}}$ is being applied.

**Definition 2.** The *safety envelope* $d_{\text{safe}}$ [1], [2] of an AV is defined as the maximum distance an AV can travel without colliding with any static or dynamic object.

A safety envelope is used to ensure (through constraints on $\mathbf{U_{A,t}}$) that the vehicle trajectory is collision-free. Production ADSs use techniques such as those in [34], [35] to estimate vehicle and object trajectories, thereby computing $d_{\text{safe}}$ whenever an actuation command is sent to the mechanical components of the vehicle. These ADSs generally set a minimum value of $d_{\text{safe}}$ (i.e., $d_{\text{safe,min}}$) to ensure that a human passenger is never uncomfortable about approaching obstacles.

**Definition 3.** The *safety potential* $\delta$ is defined as $\delta = d_{\text{safe}} - d_{\text{stop}}$. An AV is defined to be in a *safe state* when $\delta > 0$ in both lateral and longitudinal directions.[2]

### C. Fault Injection

The goal of DriveFI is to test ADSs in the presence of faults to identify hazardous situations that can lead to accidents (e.g., loss of property or life). To accomplish that goal, DriveFI includes (a) an FI engine that can modify the software and hardware states of the ADS to simulate the occurrence of faults, and (b) an ML-based fault selection engine that can find the

[2] We use the shorthand $\delta > 0$ to mean both lateral and longitudinal $\delta$s.

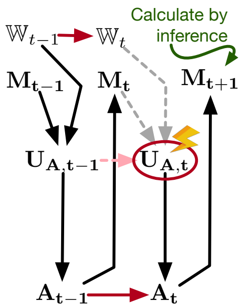


**Figure 3:** Bayesian FI.

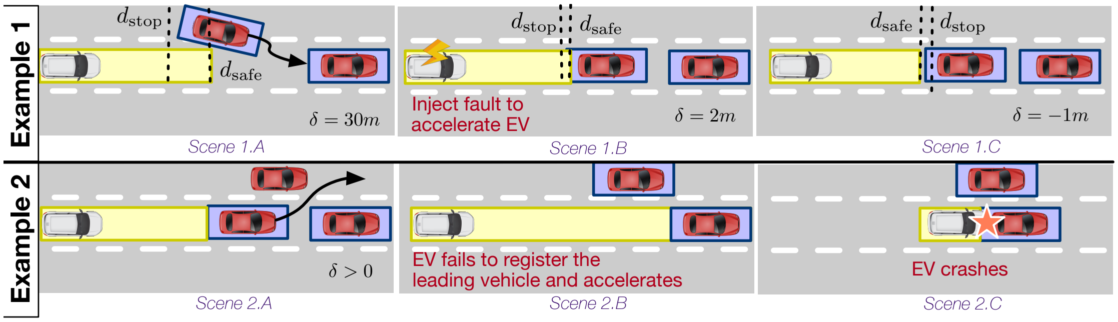


**Figure 4:** Example scenarios: (1) Targeted FI leads to hazardous conditions; (2) Real-world example with Tesla Autopilot that is similar to injected faults.

faults and scenes that are most likely to lead to violations of safety conditions and, hence, can be used to guide the fault injection. Taken together, these components of DriveFI can identify hazardous situations that lead to accidents similar to the Tesla crash described later in this section.

**Fault Model.** We assume that faults injected in DriveFI can corrupt GPU architectural state. Memory and caches (of both the CPUs and GPUs) are assumed to be protected with SECDED codes. Each injected fault is characterized by its location (in this case, its dynamic instruction count) and the injected value. The faults injected into the architectural states of these processors can manifest as *errors* in the inputs, outputs, and internal state of the ADS modules described above (i.e., $\mathbf{I_t}$, $\mathbf{M_t}$, $\mathbf{S_t}$, $\mathbf{U_{A,t}}$ and $\mathbf{A_t}$). DriveFI can directly inject errors into ADS outputs by corrupting the variables that store ADS outputs. ADS software input/output variables are ultimately stored in different levels of storage hierarchies, e.g., registers or caches. Single- or multiple-bit faults cause corruption of variables when not masked in hardware [36]. Hence, faults are being injected into these memory units, but the variables are corrupted to emulate the faults. Therefore, our fault injectors target each element in the internal ADS software state ($S\_t$), sensor inputs ($I_t$), vehicle inertial measurements ($M_t$), and actuation commands ($U_t$, $A_t$), as shown in Fig. 10. We define any error that causes safety issues for the AV as *hazardous*. For simplicity and clarity, in the remainder of the paper, we refer to both injected faults and errors as *faults*.

To build a baseline for the ML-based targeted injections, we used DriveFI to perform random injections into the GPU architectural state and ADS module outputs for two production ADS systems from NVIDIA and Baidu. In contrast to prior work [24], [30], which has reported significant SDC rates (as high as 20%) for the constituent deep-learning models (ConvNets that deal with perception: object recognition and tracking) of the ADS system, we observed that random injections rarely cause hazardous errors. These faults are masked because of the natural resilience of the ADS stack, i.e., (a) for production ADS systems that make real-time inferences at 60–100 Hz, transient faults have little chance to propagate to actuators before a new system state is recalculated; (b) the ADS system architecture is inherently resilient, as it uses algorithms like extended Kalman filtering [37] (for sensor fusion) and PID control (for output smoothing); and (c) not all driving scenes/frames are hazardous even under faults. Environmental conditions, such as the presence of other objects on the streets, are fundamental in defining the safety envelope.

**Bayesian Fault Injection.** Consider a fault $f$ that changes the value of one of the aforementioned variables. The goal of the ML-based fault injector is to find a *critical situation* that is inherently safe (i.e., $\delta > 0$) and becomes unsafe after injection of fault $f$ (i.e., $\delta_{\text{do}(f)} \leq 0$). The set of all faults $\mathbf{F}_{\text{crit}}$ in which that condition holds is defined as

$$\mathbf{F}_{\text{crit}} = \left\{ f : \delta > 0 \wedge \hat{\delta}_{\text{do}(f)} \leq 0 \right\}. \tag{1}$$

The solution to that problem requires causal and counterfactual reasoning about the behavior of the ADS under a fault. DriveFI performs that reasoning by modeling the ADS system using a *Bayesian network* (BN; shown in Fig. 3), which can capture causal relationships [38]. The BN describes statistical relationships shown by black arrows between the variables $\mathbb{W}_t$, $\mathbf{M_t}$, $\mathbf{U_{A,t}}$, and $\mathbf{A_t}$ at a time $t$, as well as relationships shown by red arrows between the variables over time. The topology of the BN is derived from the architecture of the ADS system. For example, Fig. 3 has the same graphical structure as Fig. 1. DriveFI uses the BN to calculate the maximum likelihood estimate (MLE)[3] of the value $\hat{\mathbf{M}}_{\mathbf{t+1}}$ and then uses the MLE value to calculate $\hat{\delta}_{\text{do}(f)}$ based on the kinematic model of the AV described later in §III. We use probabilistic inference over the posterior distribution of the BN to calculate

$$\hat{\mathbf{M}}_{\mathbf{t+1}} = \arg\max_{\mathbf{m}} \Pr\left[\mathbf{M_{t+1}} = \mathbf{m} \mid \text{do}(f)\right]. \tag{2}$$

The $\text{do}(\cdot)$ notation is based on the *do-calculus* defined in [38]. It marks an FI action as an intervention in the BN model. It replaces certain probabilities with constants and removes statistical conditional dependencies that are a target of the intervention (i.e., dashed lines in Fig. 3), but preserves all other statistical dependencies. We call this notion of counterfactual reasoning about the importance of a fault in performing targeted injections *Bayesian Fault Injection*.

### D. Case Studies

To explain the need for a high-efficiency FI mechanism (such as our ML-based fault injector), we discuss two examples of car accidents due to faults.

**Example 1: Hazardous Error.** Fig. 4 shows an example driving scenario in which a fault was injected into an ADS through corruption of the throttle command (which was changed from 0.2 to 0.6). The injected error led to an accident. We assume that (a) the ADS is running perception, planning, and control inference at 30 Hz, and (b) all vehicles are running on a highway with a velocity of 33.5 m/s, which is roughly the speed limit on U.S. freeways. In Scene 1A, the Ego vehicle (EV) was accelerating; however, target vehicle TV#1, operated by a human, initiated a lane change procedure, which decreased

[3] The estimated value of $x$ is denoted by $\hat{x}$.

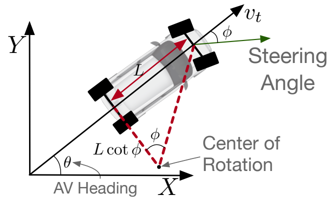


**Figure 5:** Orientation of the EV when in motion.

the safety potential $delta$ from 20 m to 2 m as shown in "Scene 1B." At that point, the Bayesian fault injector injects a fault into the throttle command, causing the vehicle to accelerate. The increase in acceleration caused the EV to become unsafe ($\delta < 0$), as shown in "Scene 1C." The EV velocity is high enough that braking, even with $a_{\text{max}}$, is not able to prevent an accident. This example shows that one needs a smart FI mechanism (such as our Bayesian-based injector) that is able to inject a fault at a precise time instant based on a run-time measurement of the safety potential to maximize the stress on the ADS and cause the EV to crash. As we argue in later sections, it is impractical (or highly difficult) to achieve the same objective using random FI.

**Example 2: Real-World Crash.** Fig. 4 shows a real-world example of a fatal accident that was shown to have been caused by a problem in Tesla Autopilot [6]. In Scene 2A, the EV followed the lead vehicle (TV#1). A few seconds later, TV#1 changed lanes (shown as Scene 2B); at that point, Autopilot decided to accelerate in order to match the allowed highway speed. However, TV#1 was behind another vehicle (TV#2), and the EV had no knowledge of TV#2; it was too late for the EV to recognize TV#2 and slow down in time to avoid an accident. While this crash was attributed to a design problem (i.e., delayed recognition) in the perception subsystem of the ADS, one can imagine that a runtime fault (that delays perception of an object) could lead to the same fatal outcome. As we show later, our Bayesian-based fault injector is able to recreate such scenarios.

## III. Bayesian Fault Injection

In this section, we describe in detail the formulation of the *Bayesian Fault Injection* approach.

### A. Kinematics-Based Model of Safety

Consider an EV moving in two-dimensional space as shown in Fig. 5. The vehicle at time $t$ has an instantaneous position $(x_t, y_t)$, speed $v_t$, heading $\theta_t$, and steering angle $\phi_t$. The equations of motion for the vehicle are

$$dx_t/dt = v_t \cos\theta_t;\ dy_t/dt = v_t \sin\theta_t;\ d\theta_t/dt = (v_t \tan\phi_t)/L, \quad (3)$$

where $L$ is the distance between the wheels of the EV [39]. Here $v_t$ and $\phi_t$ are determined by the control model for the EV. In our case, $v_t$ is defined based on the output of the ADS $\mathbf{A_t}$, i.e., $v_t = f(\zeta_t, b_t, \phi_t)$.

Note that a more complete model of the EV motion might include other dynamics, e.g., *sliding* and *skidding* of the EV's wheels. We do not add these complications to our model, as that would require us to make additional assumptions that are beyond the scope of this paper, e.g., about the EV's tires, road conditions, road banking, and weather. Similarly, we do not consider the 3-D motion of the EV, as doing so would require further assumptions about the topology of the maps (e.g., elevation) in the FI campaign. Our approach can be extended to consider those additional factors.

We can compute the maximum stopping distance $d_{\text{stop}}$ from (3) by first computing the time $t_{\text{stop}}$ taken to bring the vehicle to a complete halt, i.e.,

$$\left.\frac{dx_t}{dt}\right|_{t=t_{\text{stop}}} = 0 \text{ and } \left.\frac{dy_t}{dt}\right|_{t=t_{\text{stop}}} = 0. \quad (4)$$

$d_{\text{stop}}$ is then calculated as $[x_{t_{\text{stop}}} - x_0, y_{t_{\text{stop}}} - y_0]^T$, where $(x_0, y_0)$ is the position of the EV at the beginning of the maneuver. Closed-form solutions to the system of differential equations (3) and (4) are intractable for arbitrary control procedures (i.e., $v_t$ and $\phi_t$) and have to be solved by iterative numerical solution methods like the *Runge-Kutta methods* [40].

**The Emergency Stop Maneuver.** To simplify our analysis, we assume that the EV executes a special maneuver we call an *emergency stop* to bring the vehicle to a halt. This procedure is characterized by

$$\frac{dv_t}{dt} = -a_{\text{max}} \text{ and } \frac{d\phi_t}{dt} = 0. \quad (5)$$

That corresponds to the deceleration of the EV with the maximum deceleration to come to a halt. (5) reduces (3) to

$$d^2x_t/dt^2 = -a_{\text{max}} \sin\theta_t (d\theta_t/dt) \quad (6a)$$

$$d^2y_t/dt^2 = -a_{\text{max}} \cos\theta_t (d\theta_t/dt) \quad (6b)$$

$$\frac{d\theta_t}{dt} = \frac{\left(\sqrt{(dx_t/dt)^2 + (dy_t/dt)^2}\right)}{L} \tan\phi_0, \quad (6c)$$

where $\phi_0$ is the steering angle of the car at the beginning of the maneuver. DriveFI uses the system of equations defined in (4) and (6) to find $d_{\text{stop}}$. We use the shorthand $\mathcal{P}$ to denote the procedure (iterative numerical integration) used to compute

$$d_{\text{stop}} = \mathcal{P}(a_{\text{max}}, v_0, \theta_0, \phi_0, x_0, y_0) \quad (7)$$

from the above equations and the initial kinematic state of the EV (i.e., $v_0, \theta_0, \phi_0$, $x_0$, $y_0$) at the start of the maneuver.

Recall from §II that $\delta = d_{\text{safe}} - d_{\text{stop}}$ and that $\delta > 0$ defines the safety of the EV. The $d_{\text{safe}}$ value is assumed to be computed directly from the sensors of the EV. It is the distance to the closest object (static or dynamic) in the longitudinal or lateral path of the EV. As a result, $d_{\text{safe}}$ changes with time, and it is updated at the sensor's (e.g., LiDAR's or camera's) refresh rate. We include the boundaries of the lane in which the EV is travelling (henceforth referred to as the *Ego lane*) as a static object to be used in $d_{\text{safe}}$ computations to ensure that we capture lane violations as a safety hazard.

**Discretization.** We convert the problem of solving (7) from one that uses continuous time to one that uses a discrete notion of time. Discrete time is a natural fit for the ADS, as the control decisions are made at discrete steps that correspond to the sensors' sampling frequencies. Hence we convert time $t$ to a discrete number $k \in \mathbb{N}$ such that $t = k\Delta t$, where $\Delta t$ is the period of the sensor with the smallest sampling frequency. In the case of the DriveFI injector in DriveWorks and Apollo, that is 7.5 Hz. However, our methodology is frequency-agnostic.

### B. ML Model

The goal of a targeted fault injector is to find situations in which $\delta > 0$, but under the injection of a fault $f$ (which manifests as changes in the kinematic state of the EV) into the ADS stack, $\delta_{\text{do}(f)} \leq 0$. A solution to that problem involves speculating forward in time to after the fault has been injected, recomputing $d_{\text{stop}}$ under the fault, and then reevaluating the safety criteria for the EV. We apply an ML algorithm, which has been trained as a predictor of the EV's kinematic state, as

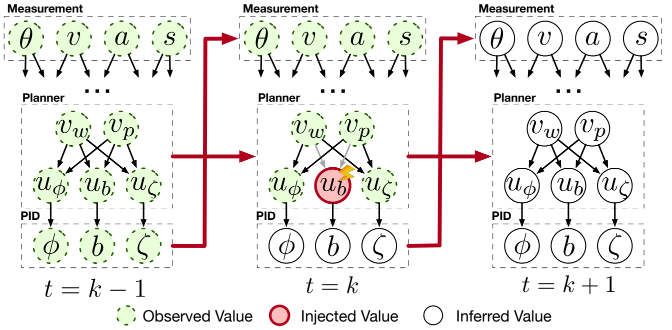


**Figure 6:** 3-Temporal Bayesian Network modeling the ADS.

the mechanism for speculation. We now describe the design of the model and its training and inference.

**The Model.** Consider a situation in which a fault is injected into the EV's ADS at time point $k$. We want to estimate the value of $d_{\text{stop}}$ at time $k+1$ when the (corrupted) actuation commands of the previous time step have been acted upon. As we showed in the previous section, we can do so using (7). However, that would require knowing the values $x_{k+1}$, $y_{k+1}$, $v_{k+1}$, $\theta_{k+1}$, and $\phi_{k+1}$ as the initial conditions to start the emergency stop maneuver. DriveFI estimates those values based on a maximum likelihood estimation over the posterior distribution of a probabilistic model that captures the components of the ADS.

DriveFI uses a Dynamic Bayesian Network (DBN) [41], specifically a 3-Temporal Bayesian Network (TBN), i.e., a DBN unfolded thrice, to model $x_{k+1}$, $y_{k+1}$, $v_{k+1}$, $\theta_{k+1}$, and $\phi_{k+1}$. This model is illustrated in Fig. 6. The core idea of DBNs is to model each point in time with a static BN and to add temporal links from one time-slice to the next (as shown by red arrows in Fig. 6). Usually all the time-points have identical BN topologies and hyper-parameter settings. BNs are directed acyclic graphs in which nodes represent random variables and arcs represent the causal connections among the variables [42]. Henceforth we will refer to each random variable in the BN is henceforth referred to as a *node* to avoid confusion with the ADS variables. Each node $x$ is associated with a probability table that provides conditional probability distributions (CPDs; $\Pr(x \mid \pi(x))$ of a node's possible value given the value of its parent nodes $\pi(x)$.

The 3-TBN model (see Fig. 6) is constructed based on the topological structure shown in Fig. 1. A detailed version of this figure for the Apollo and DriveFI ADSs is described in §IV and shown in Fig. 8. The variables in each of the ADS modules are connected in a parent-child fashion that reflects the data-flow in Fig. 8. For example, the edges between $u_\zeta$ and $\zeta$ (in Fig. 6) represent the CPD $\Pr(\zeta \mid u_\zeta)$. This is an approximation of the PID control for $\zeta$. Similarly, other components of the ADS are modeled based on their input and output variables. We assume that the nodes in the 3-TBN are described by a CPD that has the functional form

$$\Pr(x \mid \pi(x)) = \mathcal{N}(\mu_{\mathbf{x}}^{\mathbf{T}} \pi(x), \sigma_x)$$

where $\mathcal{N}$ is the normal distribution with parameters $\mu_{\mathbf{x}}$ and $\sigma_x$ (for each node $x$ in the network). That particular form of $\Pr(x \mid \pi(x))$ is chosen because (a) it has numerical stability at small probability values, which are common when dealing with rare events like faults, and (b) it simplifies the algorithm required to train the 3-TBN.

The use of the 3-TBN-based-modeling formalism is based on the implicit assumptions that (a) the EV state can be completely determined by its previous state and the observed software variables, and (b) the transition parameters from one time step to another do not change with time, i.e., the Markovian dynamic system is assumed to be homogeneous.

**Probabilistic Inference.** The maximum likelihood estimate value $\hat{v}_{k+1}$ under a manifested fault $f$ (which corresponds to setting the value of a variable in the model) is

$$\hat{v}_{k+1} = \arg\max_{v} \Pr\left(v_{k+1} = v \mid \operatorname{do}(f), \mathbf{O}_{\mathbf{k}}^{(\mathbf{f})}\right). \tag{8}$$

Given that we can execute a simulation of the EV under non-fault conditions, all variables that are not children of the injected variable can be observed to have values from the correct run. These "golden" observations are labeled $\mathbf{O}_{\mathbf{k}}^{(\mathbf{f})}$. (8) is solved by first estimating the posterior distribution of $v_{k+1}$ by using *Markov Chain Monte Carlo* methods [41] and then estimating the most likely value of $v_{k+1}$. A similar procedure can be used to compute $\hat{\theta}_{k+1}$ and $\hat{\phi}_{k+1}$. The values of $\hat{x}_{k+1}$ and $\hat{y}_{k+1}$ can then be computed using time-discretized versions of (3). Finally, from (7), we get

$$\hat{d_{\text{stop}}} = \mathcal{P}\left(a_{\max}, \hat{x}_{k+1}, \hat{y}_{k+1}, \hat{v}_{k+1}, \hat{\theta}_{k+1}, \hat{\phi}_{k+1}\right). \tag{9}$$

**Training.** The 3-TBN described above defines a probability distribution $\Pr(\mathbb{X}_{k-1}, \mathbb{X}_k, \mathbb{X}_{k+1})$, where $\mathbb{X}_k = \mathbf{M_k} \cup \mathbf{S_k} \cup \mathbf{U_{A,k}} \cup \mathbf{A_k}$. Via the BN formalism, $P(\mathbb{X}_{k-1}, \mathbb{X}_k, \mathbb{X}_{k+1})$ is defined as

$$\Pr(\mathbb{X}_{k-1}, \mathbb{X}_k, \mathbb{X}_{k+1}) = \frac{1}{Z} \prod_{x \in \mathbb{X}_k \quad 1 \cup \mathbb{X}_k \cup \mathbb{X}_{k+1}} \Pr(x | \pi(x))$$

where $Z$ is the partition function that normalizes $P$ to be a probability distribution. We use the *Expectation-Maximization algorithm* [43] to compute

$$\hat{\mu}, \hat{\sigma} = \arg\max_{\mu, \sigma} E_{\mathbb{X} \mid \mathcal{D}, \mu, \sigma} \left[\log P(\mathbb{X} \mid \mu, \sigma)\right] \tag{10}$$

where $\mathcal{D}$ refers to a training dataset that contains values of $\mathbb{X}_{k-1}$, $\mathbb{X}_k$, and $\mathbb{X}_{k+1}$ under normal operation as well as during FIs. Here, computation of $Z$ is intractable because of the combinatorially large size of $\mathbb{X}_{k-1} \times \mathbb{X}_k \times \mathbb{X}_{k+1}$. However, (8) does not require the computation of $Z$, as it is a common multiplicand to all values of the objective function.

**Training Data.** The variables in $\mathbb{X}_k$ are measured by executing the ADS in several driving scenarios in a simulator. We describe the setup of this simulator in §IV. Simply capturing the data under normal operation is not sufficient to capture abnormalities created in the ADS state because of faults. Therefore, in addition to running driving scenarios without faults, we run the driving scenarios while injecting random faults (i.e., the baseline described in §II) one at a time. The FI campaign that corresponds to the training data is described in §V. We recreate the process of injecting a fault into a uniformly randomly selected scene 20 to 50 times for each fault. The reason for varying the number of faults is that some variables (such as $\zeta$, $b$, and $\phi$) exhibit all possible values during simulated runs with no injections, while others, such as stateful variables, simply do not vary naturally.

**Fault Injection.** The computation of $\mathbf{F}_{\text{crit}}$ (from (1)) is done offline for every frame in every driving scene. The FI procedure executes as follows (see Fig. 7):

- For each driving scenario, a non-fault-injected "golden" execution of the simulation is performed. At each instant $k$,

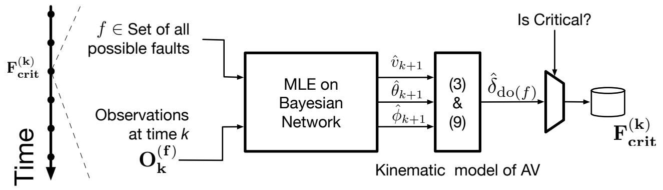

$\mathbf{F}_{\text{crit}}^{(\mathbf{k})}$
$f \in$ Set of all possible faults
Observations at time $k$
$\mathbf{O}_{\mathbf{k}}^{(\mathbf{f})}$
Time
MLE on Bayesian Network
$\hat{v}_{k+1}$
$\hat{\theta}_{k+1}$
$\hat{\phi}_{k+1}$
(3) & (9)
Is Critical?
$\hat{\delta}_{\text{do}(f)}$
$\mathbf{F}_{\text{crit}}^{(\mathbf{k})}$
Kinematic model of AV

**Figure 7:** BN MLE inference is executed offline for every simulated time point to find the set of critical faults.

the variables in $\mathbb{X}_k$ are measured and stored.

- These "golden" values of $\mathbb{X}_k$ are stepped through with (9) to build $\mathbf{F}_{\text{crit}}^{(\mathbf{k})}$ for every scene/frame, based on (1).
- An FI campaign is carried out on the simulated EV to execute faults in $\bigcup_k \mathbf{F}_{\text{crit}}^{(\mathbf{k})}$ one frame and one fault at a time.

## IV. The ADS Architecture & Simulation

### A. AI Platform

An AV uses ADS technology to support and replace a human driver for the tasks of controlling the vehicle's steering, acceleration, and monitoring of the surrounding environment (e.g., other vehicles/pedestrians) [44]. The ADS architecture consists of five basic layers [4], discussed below:

**Sensor Abstraction Layer** (❶ in Fig. 8): The sensor abstraction layer is responsible for preprocessing of input data, noise filtering, gains control [45], tone-mapping [46], demosaicking [47], and extraction of regions of interest, depending on the sensor type. An ADS supports a wide range of sensors, such as Global Positioning System (GPS), Inertial measurement unit (IMU), sonar, RADAR, LiDAR, and camera sensors. Our experiments only use two cameras (fitted at the top and front of the vehicle) and one LiDAR.

**Perception Layer** (❷ in Fig. 8): The sensor abstraction layer feeds data into the perception layer, which uses computer vision techniques (including deep learning [48]) to detect static objects (e.g., lanes, traffic signs, barriers) and dynamic objects (e.g., passenger vehicles, trucks, cyclists, pedestrians) present in a driving scenario.

The object detection algorithm performs several tasks (e.g., segmentation, classification, and clustering). It uses all the sensor data separately and then merges the data using sensor fusion algorithms (e.g., extended Kalman filtering [37], [49]). The fusion algorithm provides software-level data redundancy for object detection. Use of HD maps and the localization module enables the ADS to predetermine the location of specific static objects, such as traffic lights, further improving the confidence in obstacle-detection tasks.

The perception layer is also responsible for temporal tracking of objects and lanes. Tracking is necessary to ensure that an object does not suddenly disappear from a frame because of misclassification or a failure to detect anything. Thus, sensor fusion and tracking provide spatial and temporal redundancy in the perception layer of the software. After accurate determination and tracking of objects and lanes are completed, the perception layer calculates various useful metrics such as "closest in path obstacle" (CIPO) and "tailgating distance" for each object. Such association of an object with measured or inferred metrics (e.g., CIPO and tailgating distance) is defined as the *world model* of the AV.

**Localization Layer** (❸ in Fig. 8): The localization module is responsible for aggregating data from various sources to locate the autonomous vehicle in the world model. Localization in the world model can be done using a GPS sensor or by using camera/LiDAR inputs. The work described in this paper uses only camera/LiDAR along with maps to enable localization (i.e., it does not use GPS).

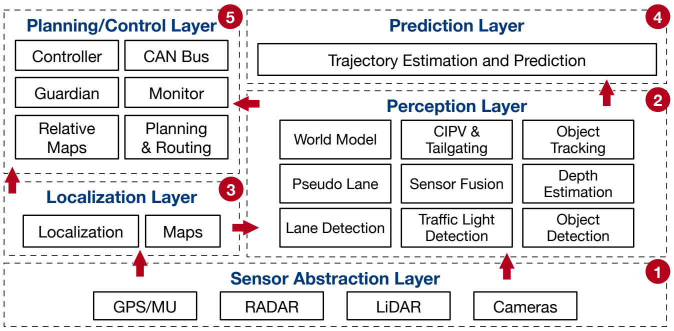


**Figure 8:** ADS architecture.

**Prediction Layer** (❹ in Fig. 8): The prediction layer is responsible for generating trajectories for detected objects by using information from the world model (e.g., positions, headings, velocities, accelerations). As a result, it can probabilistically identify obstacles in an AV's path [50].

**Planning & Control Layer** (❺ in Fig. 8): The planning and control layer is responsible for generating navigation plans based on the origin and destination of the EV and for sending control signals (actuation, brake, steer) to the AV. The "Routing module" generates high-level navigation information based on requests. The Routing module needs to know the routing start point and routing end point, in order to compute the passage lanes and roads. The "Planning module" plans a safe and collision-free trajectory by using localization output, prediction output, and routing output. The "Control module" takes the planned trajectory as input and generates the control command to pass to the CAN Bus, which passes the information to the AV's mechanical components. The surveillance system monitors all the modules in the vehicle, including hardware. The "Monitor module" receives data from different modules and passes them on to a human-machine interface for the human driver to view to ensure that all the modules are operating normally. In the event of a module or hardware failure, the monitor triggers an alert in the "Guardian module," which then chooses an action to be taken to prevent an accident.

### B. Simulation Platform

This paper uses Unreal Engine (UE) based simulation platforms (Carla [51] and DriveSim [52]) that are capable of simulating complex urban and freeway driving scenarios by using a library of urban layouts, buildings, pedestrians, vehicles, and weather conditions (e.g., sunny, rainy, and foggy). The simulation platforms are capable of generating sensor data at regular intervals (from cameras and LiDARs) that can be fed to the ADS platform. A driving scenario consists of 500 scenes in DriveAV or 2400 scenes in Apollo in which the EV travels from a fixed starting point on the road to a fixed destination point. A scene in a driving scenario is a representation of the physical world at the simulation epoch and corresponds to a camera frame. Fig. 9 illustrates scenes from three freeway (DS1–DS3) and three urban (DS4–DS6) driving scenarios used in this study. DS1–3 are controlled by DriveAV in DriveSim, and DS4–6 by Apollo in Carla. In these scenarios an EV and a few UE-controlled TVs/pedestrians

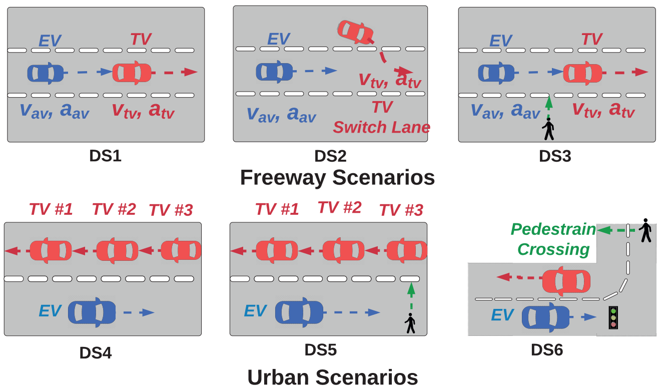


**Figure 9:** Driving scenarios supported by simulation engine.

are placed in urban and freeway roads, driving at different velocities/accelerations and separated by some distance. The EV is expected to execute driving maneuvers in each of these settings. The scenarios represent the most common driving cases encountered by humans on a daily basis. In DS1–DS6, the Ego vehicle does not switch lanes, there is no other vehicle trailing the Ego vehicle, and the Ego vehicle is in a safe state.

### C. Hardware Platform

The NVIDIA DriveAV ADS was designed for the NVIDIA AGX Pegasus platform [53], which consists of two Xavier SoCs and two discrete GPUs, but is also supported on a development platform based on an x86 CPU and a GPU. For our experiments, we used the development platform and its utilities to facilitate the creation of the DriveFI tool. The Apollo ADS is supported on the Nuvo-6108GC [54], which consists of Intel Xeon CPUs and NVIDIA GPUs. We use Apollo on an x86 workstation with two NVIDIA Titan Xp GPUs.

## V. DriveFI Architecture

The software architecture of DriveFI is shown in Fig. 10. DriveFI leverages the existing tools to simulate driving scenarios and control the EV in simulation by using an AI agent (which is provided by Apollo or DriveAV). The scenario manager coordinates the simulator and AI agent to run a driving scenario and monitor the state of the software as well as the safety of the EV. DriveFI is bundled with a campaign manager that takes an XML configuration file as input to select a fault model, software or hardware module sites for FI, the number of faults, and a driving scenario. The campaign manager uses the specified configuration to (a) profile the ADS workload, (b) generate a fault plan[4], and (c) inject one or more transient faults per run into the ADS system. Based on the values in the configuration file, the campaign manager runs a specified number of golden simulations, profiles the ADS while running a driving scenario, and runs a specified number of experiments that inject one or more faults at a time based on the generated fault plan. The “Event-driven synchronization” module helps coordinate among all the toolkits (the UE-based driving scenario simulator, monitoring agents, campaign manager, fault injectors, and AI agent).

We built DriveFI to characterize error propagation and masking (a) in computational elements, (b) in the ADS, and (c) in vehicle dynamics and traffic. Low-level circuit, micro-architectural, and RTL faults manifest as architectural-state faults in computational elements. The architectural-state faults that do not get masked manifest as errors in the internal state of the ADS modules, and the errors that do not get masked in the module propagate to the output of the module. Finally, errors that are not masked in any of the modules manifest as actuation command errors that are sent to the AV. Therefore, to mimic faults and errors, we built two fault injectors: (a) a GPU fault injector (GI; see Section V-A) capable of injecting faults into the GPU architecture state to reveal the propagation of GPU faults to the ADS state, and (b) a source-level fault injector (SLI, see Section V-B) capable of injecting faults to corrupt ADS software variables. Corruption of the final output (actuation values $\zeta, b, \theta$) of the ADS helps us to measure the resilience associated with vehicle dynamics and traffic. Thus, our approach aids in the measurement of fault and error masking/propagation at different levels and the corresponding impact on the safety of the AV.

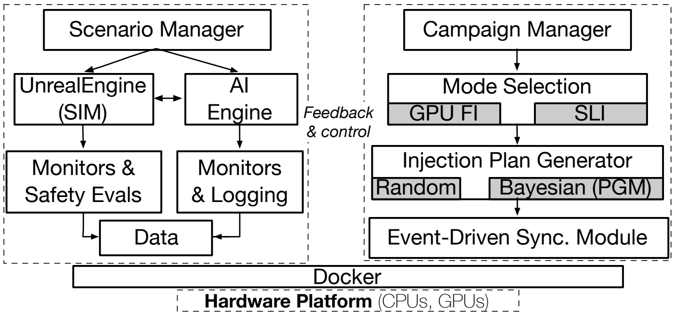


**Figure 10:** DriveFI architecture.

DriveFI is bundled with a campaign manager that takes an XML configuration file as input to select a fault model, software or hardware module sites for FI, the number of faults, and a driving scenario. The campaign manager uses the specified configuration (a) to profile the ADS workload, (b) to generate a fault plan, and (c) to inject one or more transient faults per run into the ADS system. For this paper, we developed an “event-driven synchronization” module that coordinates among all the toolkits (the UE-based driving scenario simulator, monitoring agents, campaign manager, fault injectors, and AI agent).

### A. Injecting into Computational Elements: GPU Fault Models

We consider transient faults in the functional units (e.g., arithmetic and logic units, and load store units), latches, and unprotected SRAM structures of the GPU processor. Such transient faults are modeled by injecting bit-flips (single and double) in the outputs of executing instructions. If the destination register is a general-purpose register or a condition code, one or two bits are randomly selected to be flipped. For store instructions, we flip a randomly selected bit (or bits) in the stored value. Since we inject faults directly into the live state (destination registers), our fault model does not account for various masking factors in the lower layers of the hardware stack, such as circuit-, gate-, and micro-architecture-level masking, as well as masking due to faults in architecturally untouched values. The GI employs an approach similar to that of SASSIFI [23] and includes a profiling pass and fault-injection plan generation. We do not consider faults in cache, memory, and register files, as they are protected by ECC.

### B. Injecting Faults into ADS Module Output Variables

The goal of SLI (Source-Level Injection) is to corrupt the internal state of the ADS by modifying ADS module output

[4] A fault plan specifies which instruction/variable to corrupt, the corruption time, and the corruption value.

**Table I:** Examples of SLI-supported ADS module outputs.

| **FI Target (Output Variables)** |
|---|
| **Path Perception Module** |
| `lane_type, lane_width` |
| **Object Perception Module** |
| `camera_object_distance, camera_object_class,`<br>`lidar_object_distance, lidar_object_class,`<br>`sensor_fused_obstacle_distance, sensor_fused_obstacle_class` |
| **Planning & Control Module** |
| `vehicle_state_measurements` $(pos, v, a)$, `obstacle_state_measurements` $(pos, v, a)$, `actuator_values` $(\zeta, b, \phi)$, `pid_measured_value`, `pid_output` |

variables (hence, the input variables of another module) of the ADS components. SLI is implemented as a library that is statically linked to the ADS software; however, its use requires source-code modification and recompilation of the ADS software. We did not observe any noticeable runtime difference between SLI-linked ADS and non-SLI ADS. In this work, we manually identified the software variables that store the outputs of ADS modules that play a critical role in inferring the actuation commands of the EV. Source-code modification is required in order to mark the output variable and invoke the corresponding module injector to get a corrupted value by using the fault model provided in the XML config file. In Table I, we show some of the variables from each of the ADS modules (see Fig. 8) that were targeted using SLI.

The fault models supported by SLI that corrupt one or more software output variables in the $k^{\text{th}}$ scene (chosen uniformly and randomly over all scenes of a driving scenario) are specified by (a) a number of faults (i.e., a number of consecutive scenes to be injected), and (b) the fault location. A *single fault* in SLI-based experiments is the corruption of a single output variable of an ADS module. In the following, we define these SLI-supported fault models.

**1-Fixed**. A single fault is injected at the $k^{\text{th}}$ scene of a given ADS software module output. Across experiments, a constant value is used to corrupt the given ADS software module output. There are a total of 41 “1-Fixed” fault types, each defined by (a) the ADS module output, and (b) the corruption value. The bounded continuous outputs are corrupted to maximum or minimum possible value for those outputs, For example, to inject into brake actuation output, SLI uses a maximum brake value of 1.0 or a minimum brake value of 0.0. Unbounded continuous output values (e.g., $v$, $a$, and $pos$) are corrupted to double or half of the current output value[5]. For categorical output variables the output value is corrupted to one of the categorical values; e.g., the object/obstacle class can be corrupted to “do not care/disappear,” “pedestrian,” “vehicle,” and “cyclist.”

**M-Fixed**. $m$ faults are injected into a given set of ADS software module output starting at scene $k$, and continues to inject faults into the ADS software module output until scene $K + m$. $m$ is chosen uniformly and randomly between 10 and 100. The range selected for $m$ is large enough to support study of a threshold value for a number of consecutive frames/scenes that must be injected to cause a hazardous situation. Again, there are 41 “M-Fixed” fault types.

**1-Random**. A single fault is injected at the $k^{\text{th}}$ scene in a uniformly and randomly chosen set of ADS module output. The injected fault value is also chosen uniformly and randomly from the range of values of the selected ADS module output.

**Table II:** Fault injection experiments.

| **Campaign** | **Target module** | **#Faults/Experiment** |
|---|---|---|
| 1-GPU-all | All GPU kernels | 1 |
| 1-RANDOM | All software module outputs | 1 |
| 1-Fixed_throttle_max | Actuator - throttle | 1 |
| 1-Fixed_brake_max | Actuator - brake | 1 |
| 1-Fixed_Steer_max | Actuator - steer | 1 |
| 1-Fixed_obstacle_rem | Perception - obstacle disappear | 1 |
| 1-Fixed_obstacle_dist | Perception - obstacle distance | 1 |
| 1-Fixed_lane_rem | Perception - lane disappear | 1 |
| M-Random | All software module outputs | 10–100 |
| M-Fixed_throttle_max | Actuator - throttle | 10–100 |
| M-Fixed_brake_max | Actuator - brake | 10–100 |
| M-Fixed_Steer_max | Actuator - steer | 10–100 |
| M-Fixed_obstacle_rem | Perception - obstacle disappear | 10–100 |
| M-Fixed_obstacle_dist | Perception - obstacle distance | 10–100 |
| M-Fixed_lane_rem | Perception - lane disappear | 10–100 |
| 1-PGM | All software modules | 1 |

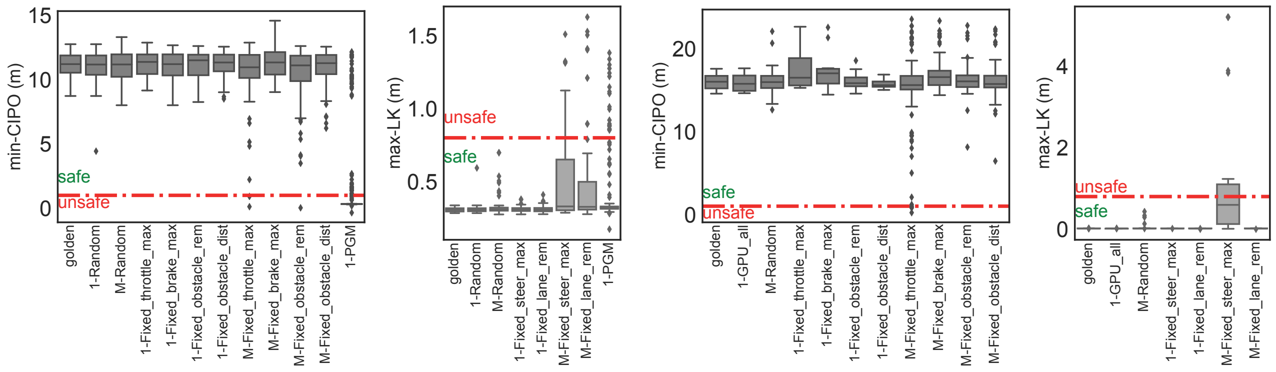
**M-Random**. $m$ faults are injected in a set of randomly chosen ADS software module output starting at scene $k$, and continues to inject faults in the ADS software module output until scene $K + M$. $m$ is chosen uniformly and randomly between 10 and 100. In this case, both the ADS module and the corruption value are selected uniformly and randomly.

## VI. Results

In this section, we characterize the impact of fault and error injection on the safety of the EV. In our work, we use a UE-based simulator to study three freeway driving scenarios (DS1–DS3) and three urban driving scenarios (DS4–DS6). DS1–DS3 were controlled by DriveAV, whereas DS4–DS6 were controlled by Apollo. The safety of the EV at any given scene is verified by calculating the CIPO (the closest in path obstacle) and LK distance(lateral distance from the center of the lane). A safety hazard occurs when $d_{min} < 1.0$ m in the longitudinal direction, which corresponds to less than 1.0 m of minimum distance from CIPO, or when the EV crosses the Ego lane, which corresponds to a 0.80 m displacement from the center of the lane. Hence, the minimum CIPO distance (min-CIPO) and maximum LK distance (max-LK) across all scenes characterize the safety hazard for the entire simulation.

Because of space restrictions, without any loss of generality, we limit our discussion to DS1, in which the EV was controlled by DriveAV, and DS6, in which the EV was controlled by Apollo. Figs. 11a–11d show the boxplots of min-CIPO and max-LK for Apollo (DS6) and DriveAV (DS1), respectively, across all fault injection experiments and golden runs. These experiments are summarized in Table II. A boxplot shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. The boxplot shows the quartiles of the dataset, while the whiskers extend to show the rest of the distribution (maximum and minimum samples), except for points that are determined to be outliers [55]. To understand the simulation and safety characteristics of the driving scenarios, we ran 50 end-to-end simulations for each scenario without any injection. These runs are called *golden runs*. The golden runs serve as a reference against which we compare injected simulation runs in the rest of the paper. The median min-CIPO and max-LK distances are 16 m (see “golden” in Fig. 11c) and 0.019 m (see “golden” in Fig. 11d) for DriveAV, and 11.19 m (see “golden” in Fig. 11a) and 0.31 m (see “golden” in Fig. 11b) for Apollo. None of the golden runs resulted in safety hazards.

[5] We limit ourselves to corruption of the outputs to double or half, as otherwise the ADS may detect the injected faults as errors.



**(a)** Apollo min-CIPO **(b)** Apollo max-LK **(c)** DriveAV min-CIPO **(d)** DriveAV max-LK

**Figure 11:** Fault/error impact characterization using FI campaigns. (a) & (b) use DS6; (c) & (d) use DS1.

### A. *GPU-level Fault Injection*

We conducted 800 GPU-level FI experiments for each driving scenario (DS1, DS2, DS3) in DriveAV. The min-CIPO and max-LK of DS1 simulated in DriveAV are labelled as "1-GPU" in Fig. 11c and Fig. 11d, respectively. We conducted only 800 GPU-level FI experiments per scenario because we did not observe any safety violations during the runs, and running more experiments would have been prohibitively expensive (2.7 days per driving scenario, 800*5 minutes/FI). In FI experiments labelled "1-GPU_all", faults were chosen uniformly randomly from across all dynamic instructions in the ADS. We did not conduct any GPU FI experiments on Apollo because of a CUDA driver version mismatch between GI and Apollo. Resolving the issue would have required vendor support and fixes. From Fig. 11c and Fig. 11d, we can observe that the EV is always safe, even after FI, and that the distribution is similar to the one in the golden case.

**Fault propagation and masking in GPUs.** Across all GPU-FI experiments on the DS1–DS3 driving scenarios, representing a total of 2400 FI experiments, 1.9% of injected faults led to silent data corruption (i.e., caused corruption of actuation outputs which are the final outputs of the ADS module), and 0.02% led to object misclassification errors[6]. None of the object misclassification errors resulted in actuation output corruption. Our results indicate that the perception module (which is responsible for object detection and classification) is more resilient than other ADS modules. The reason is that the perception software takes advantage of sensor fusion (i.e., redundancy in sensing devices can compensate for a fault of a single sensor). Across all driving scenarios, the SDCs did not result in any EV safety breach.

7.35% of faults resulted in detectable uncorrectable errors (DUEs) that led to ADS software crashes (61%) or hangs (39%). The ADS is equipped to handle detectable errors and take corresponding corrective or safety measures. Although DUEs are more common than SDCs, it is expected that systems can recover from such faults via the backup/redundant systems.

**Errors persist for multiple frames**. In 2% of the misclassification error cases (recall that 0.02% of GPU-level FIs led to misclassification errors), ADS perception module outputs were incorrectly classified for more than one frame, i.e., the impact of the injected fault persisted for more than one frame. In our data, we observed misclassification of objects for up to eight continuous frames. In those cases, errors did get masked eventually because of the temporal nature of the ADS platform. For example, ADS is fed with new sensor data at regular intervals, e.g., 7.5 times per second in our study. This observation suggests the need for more thorough study of fault masking and propagation in ADSs at the software level to handle cases in which faults persist for more than one frame.

### B. *Source-level Fault Injections*

We observed in the previous section that the ADS was able to compensate for injected transient faults. To further understand the ADS platform's susceptibility to faults and its robustness in the case of persistent errors, we conducted targeted FI with SLI to inject one or more faults directly into the ADS module outputs. We conducted 84 SLI-based FI campaigns for each driving scenario (scenarios 1–3 in DriveAV and 4–6 in Apollo). Of the 43 campaigns, 1 corresponded to "1-Random," 1 corresponded to "M-Random," 41 corresponded to 41 fault types under "M-Fixed," and 41 corresponded to 41 fault types under "M-Random." Labels are shown in Fig. 11.

**Robustness of the ADS to single and multiple faults.** The ADS platform was found to be robust to injection of a single fault ("1-Random" campaign). To understand the robustness to persistence of fault-generating multiple random errors, we conducted FI campaigns on driving scenarios by using "1-Random" and "M-Random" fault models. The distributions of min-CIPO and max-LK for "M-Random" were found to be statistically different from those in the golden runs for Apollo (see "M-Random" in Fig. 11a and Fig. 11b) and DriveAV (see "M-Random" in Fig. 11c and Fig. 11d). For both "1-Random" and "M-Random" campaigns, none of the injected faults led to a hazardous driving situation; however, the ADS safety was found to be more vulnerable [7] to the "M-Random" fault model (especially for lane keep functionality). For example, the minimum min-CIPO observed across all injections decreased from 8.7 m to 8.0 m, and max-LK increased from 0.34 m to 0.7 m for Apollo. Similarly in DriveAV, min-CIPO increased from 15.2 m to 12.6 m, and max-LK decreased from 0.024 m to 0.43 m.

**Robustness of the ADS modules to single and multiple faults.** A persistent fault within the component of the ADS module continuously generates errors for the corresponding module. We tested the robustness of the ADS to a faulty

[6] *Object misclassification* refers to incorrect classification of an object, e.g., a pedestrian may be recognized as a vehicle.

[7] The AV came closer to the other vehicle/pedestrian compared to when no fault was injected.

module by subjecting one of the chosen module outputs to multiple faults. In these campaigns, we used "1-Fixed" and "M-Fixed" fault models. There are a total of 41 fault types for "M-Fixed" and "1-Fixed" fault types (e.g., "throttle max," "obstacle removal," and "lane removal"). We discuss the results of only select campaigns because of lack of space. The selected campaigns (shown in Fig. 11) included (a) actuation module output corruption (in which the brake, throttle, and steering were all changed to the "max" allowed value); (b) sensor fusion output corruption (in which the obstacle class was changed to "disappear" and the distance that could be considered in trajectory planning was changed to "max"); and (c) lane output corruption (in which the lane type was changed to "disappear"). The FI experiments that led to safety breaches appear as data points below the red line for min-CIPO and above the red line for max-LK. Clearly, none of the FI campaigns conducted under the "1-Fixed" fault model led to safety hazards, but few were observed for "M-random" FI campaigns. We rank ADS modules by their module vulnerability factor (MVF), which we calculate by finding the percent of simulations that resulted in either (a) a min-CIPO distance less than the minimum min-CIPO distance across the golden runs, or (b) a max-LK distance maximum more than the max-LK distance across golden simulation runs. Using that method, we find that the "steer angle" (MVF=46%), "lane classification" (MVF=43%), "obstacle classification" (MVF=10%), and "throttle" (MVF=7%) are most vulnerable for Apollo, whereas for DriveAV we find the same components to be vulnerable except for "lane classification" and "obstacle removal".

The higher resilience of "lane classification" and "obstacle removal" in DriveAV can be attributed to the free-space detection module (not present in Apollo) and the scene attributes. The free-space detection module helps the DriveAV EV to detect drivable space (using a dedicated DNN network tasked with finding drivable space) even if the object is misclassified or its attributes (such as distance and velocity etc.) are corrupted. The free-space detection module ensures safety without requiring complete replication of obstacle detection and classification modules. The masking of faults in both modules can also be attributed to obstacle registration and tracking in the world model that helps track the obstacle over time.

**Compensation in ADS**: An ADS automatically compensates for any change in EV state (i.e., $\theta, v, a, s$) that leads to an unsafe state caused by one or more faults/errors. It does so by issuing actuation commands that bring the EV to a safe state. For example, the EV may compensate for an increased $v$ by braking ($b$), a decreased $v$ by throttling ($\zeta$), or a change in heading angle by steering ($\phi$). Fig. 12 shows throttle ($\zeta$) values for golden and injected runs (in the left subfigure) and compensation achieved by braking (in the right subfigure) for an FI experiment in which $\zeta$ was corrupted in 30 consecutive frames/scenes. Compensation at time step $K$ is calculated as the difference between the cumulative sums of "brake" values observed at time step $K$ in the injected run and in the golden runs. The injection leads to an increase in the velocity of the vehicle, which is compensated for by braking. In the right subfigure in Fig. 12, we show that the compensation increases until time step $K = 232$ to undo the effects of multiple faults, and then flattens out as the brake values in the golden run and faulty run (i.e., run with fault injection) become equal. We observed similar compensation behavior for the faults injected into brake and steer values.

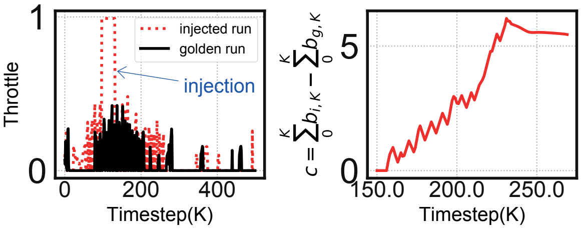


**Figure 12:** Impact of 30 continuous faults on $\zeta$ in DriveAV. Left subfigure shows $\zeta$ for a golden simulation (in black) and an injected simulation (in red). Right subfigure shows compensation $c$.

The ability of an ADS to compensate for injected faults depends on the number of faults and the time of injection. The outlier data point below the red line in Fig. 11a for "M-Fixed_throttle_max" corresponds to 30 consecutive frames/scenes injected with faults into $\zeta$ values. In this FI experiment (not shown in Fig. 12), the vehicle was not able to compensate for the injected faults, as the faults were injected at $K = 400$ and there was not sufficient time for the vehicle to stop, i.e., the EV reached an unsafe state at the end of the injections. In Apollo, only 20 injected faults into $\zeta$ values led to unsafe states. *Persistent errors have significant impact on the EV's state, and the ADS's ability to compensate for the impact of errors depends on the time and location of FIs.*

### C. Results of Bayesian FI-based injections

In our FI campaigns thus far, hazardous driving conditions (accidents and lane violations) were created only when multiple faults had been injected into the ADS (i.e., multiple consecutive frames/scenes had been injected). However, in the real world, it is more likely that a single fault will occur, and therefore it is important to find conditions under which a single fault can lead to hazardous driving conditions. One way to approach the problem of finding all such single faults (i.e., critical faults) is to inject every single fault while running a driving scenario in a simulator. That approach, however, would be prohibitively costly and is infeasible in practice. For example, an exhaustive search to find which of the 41 fault types under the "1-Fixed" fault model will lead to safety hazards would have taken 272 days[8] [9] in our simulation platform . Another way to find critical faults is to inject faults uniformly and randomly. However, the results from GPU hardware-level FI (see §VI-A) and ADS software module-level FI (see §VI-B) suggest that we need a smart FI method capable of identifying hazardous situations in driving scenarios and using them to guide FI experiments. A fault injector based on such a method would inject a fault when the ADS is most vulnerable (i.e., the fault is likely to propagate to actuators) and in such a way that the ADS cannot compensate for the fault. The Bayesian fault injector is able to find a *critical situation* that was inherently safe (i.e., $\delta > 0$) but became unsafe after injection of fault $f$ (i.e., $\delta_{\mathrm{do}(f)} \leq 0$). We have shown the effectiveness of Bayesian FI by injecting faults into driving scenarios DS4-DS6 controlled by Apollo.

**Effectiveness of Bayesian FI.** When we used Bayesian FI, 82% of injected faults resulted in hazards. (95% of the hazards were accidents involving a pedestrian, and 5% were

[8] $^{615\text{ days}}/_{DS} = {}^{9\text{ min}}/_{DS} * 41$ fault types $*$ 2400 scenes.

[9] Note that traditional FI is sampling-based, so 615 days represents the worst case of enumeration of all faults.

lane violations.) Bayesian FI selects one of the 41 fault types of the "1-Fixed" fault model, and uses SLI to inject a single fault into an ADS module output variable. Recall that in the"1-Fixed' fault model, the fault location (i.e., the ADS module output variable) and corruption value are defined by the fault type. In comparison, none of the random single FIs led to safety hazards. The Bayesian FI results are marked as "1-PGM" in Fig. 11a and Fig. 11b. All data points below the red line in Fig. 11a correspond to collisions, and all data points above the red line in Fig. 11b correspond to lane violations. The median min-CIPO distance was 0.32 m, which is significantly less than the 11.19 m median value for golden runs. Although the median max-LK value did not change for the "1-PGM" campaign compared to golden runs, 5% of the hazards were due to lane violations.

**Mining critical faults and critical scenes.** As discussed before, injection of all fault types under the "1-Fixed" fault model of SLI would be prohibitively expensive. Bayesian FI helped us find all critical faults $|F_{crit}|$ for every scene and mine driving scenes that are more susceptible to faults. The critical faults mined by Bayesian FI can help designers understand the weaknesses of the system and corner cases under which a fault may lead to hazards, whereas the critical scenes can be used by designers to inject random faults (using GI or SLI) only in those scenes to help them understand the architecture vulnerability factor (AVF). We believe that the mining of critical scenes by Bayesian FI will have wider applicability beyond our FIs here. Combination of results from a range of FI experiments to create a library of scenes will help manufacturers develop rules and conditions for AV testing and safe driving. Table III gives summary statistics of mined critical faults and scenes in the driving scenarios (DS4–6). A total of 561 faults were found to be critical across DS4–6. Upon inspecting the mined critical faults, we found that the top 3 most susceptible ADS module outputs for vehicle collision are the throttle value (24% of 561 critical faults), the PID controller input (18%), and the sensor-fusion obstacle class value (15% of 561 critical faults). ADS module outputs targeted by Bayesian FI for creating lane violations are the (a) lane type value (2% of 561), (b) throttle (1.4%), and (c) steer (1.4%). 56% of the fault types were never used by Bayesian FI; for example, Bayesian FI never injected into the output of camera-sensor object classification module.

For DS4, we did not find any critical scene or error. That was expected, as there was no trailing or leading vehicle around the EV in our driving scenarios. All the vehicles were in the other lane following a completely different trajectory, and one fault in this case would not be sufficient to make the EV cross into the adjacent lane. For DS5, 0.88% of the scenes and 0.20% of the faults were found to be critical. The critical scenes in this case correspond to a scene in which (a) the object (i.e., pedestrian) is first registered into the world model and (b) the EV then starts braking. In case of (a), the Bayesian FI chooses to remove the obstacle (e.g., by removing the obstacle, or misclassifying the object), and in the case of (b), the Bayesian FI chooses to accelerate the vehicle (e.g., by corrupting PID outputs or planner outputs). For DS6, we observed that 1.96% of the scenes and 0.36% of the faults were critical. We made a similar observation for DS5. However, in addition, we found the EV to be susceptible to faults around turns. Bayesian FI in those cases chooses faults that correspond to a disappearing lane or steering value corruptions. The EV tends to follow the lead vehicle when the lane markings are missing. However, in turns for which there is no lead vehicle to follow, such errors become critical. *It is worthwhile to note that Bayesian FI was able to mine critical faults and scenes in 4 hours, and took approximately 54 hours to simulate all the extracted faults in the simulator.*

**Table III:** Summary of PGM-based fault injection.

| Driving scenario | Crit. scenes % | Crit. faults % | Hazard rate |
|---|---|---|---|
| DS4 (2400 scenes) | 0 | 0.0 | 0.0 |
| DS5 (2400 scenes) | 0.88 | 0.20 | 0.36 |
| DS6 (2400 scenes) | 1.96 | 0.36 | 0.20 |

[1] Total faults (TF) in the "FIXED" fault model = #scenes/DS * #error types = 98400/DS
[2] Critical scenes % = #scenes in which critical faults were found by #scenes/DS
[3] Critical faults % = (Critical faults mined by Bayesian FI)/TF

## VII. Related Work

AV research has traditionally focused on improvement ML/AI techniques. However, as models are deployed at large scale on computing platforms, the focus changes to assessment of the resilience and safety features of the compute stack that drives the AV. Assessment of the safety and resilience of AVs requires robust testing techniques that are scalable and directly applicable in real-world driving scenarios. It is not scalable or practical to base a safety argument solely on statistical measures such as a billion miles on roads, or on simulations done on platforms such as CARLA [51] or Open Pilot [5], [56], [57]. Testing the robustness of an ADS has proven to be challenging and mostly ad hoc or experience-based [17]. In particular, to test the functionality and design of the hardware and software components of an ADS, current methods rely on injection of invalid or perturbed inputs [28], [31], [32] or faults and errors [24], [31], [58] into an ADS in simulation or ADS components, and accrual of millions of miles on roads [59].

However, these methods are not scalable because (a) they lack simulated or real datasets that would represent all kinds of driving scenarios [56]; (b) it would take billions of miles of driving to add functionality or do a bug fix, in order to drive statistical measures [60]; (c) they are restricted to DNNs [24], [57], [61]–[63] and sensors [31], [32], even when DNNs form only a small part of the whole ecosystem; and (d) once the easy bugs have been fixed, finding rare hazardous events would be exponentially more expensive, as faults might manifest only under specific conditions (e.g., a certain software state).

## VIII. Conclusion

In this work we present DriveFI, a fault injection tool, along with methodologies to empirically assess the fault propagation, resilience, and safety characteristics of the ADS, as well as to generate and test corner-case failure conditions. DriveFI incorporates Bayesian and traditional FI frameworks which work in tandem to accelerate finding of the safety-critical faults.

## Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CNS 15-45069. We thank K. Atchley, J. Applequist, K. Saboo, and S. Cui. We would also like to thank NVIDIA Corporation for software access and equipment donation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF and NVIDIA.

# REFERENCES

[1] S. M. Erlien, "Shared vehicle control using safe driving envelopes for obstacle avoidance and stability," Ph.D. dissertation, Stanford University, 2015.

[2] J. Suh, B. Kim, and K. Yi, "Design and evaluation of a driving mode decision algorithm for automated driving vehicle on a motorway," *IFAC-PapersOnLine*, vol. 49, no. 11, pp. 115–120, 2016.

[3] Nvidia, "nvidia drive | nvidia developer," https://developer.nvidia.com/driveworks.

[4] Baidu, "Apollo Open Platform," http://apollo.auto, Accessed: 2018-09-02.

[5] CommaAI, "OpenPilot: Open source driving agent," https://github.com/commaai/openpilot, Accessed: 2018-09-12.

[6] S. Alvarez, "Research group demos why Tesla Autopilot could crash into a stationary vehicle," https://www.teslarati.com/tesla-research-group-autopilot-crash-demo/, June 2018.

[7] T.S., "Why Uber's self-driving car killed a pedestrian," The Economist May 29, 2018 https://www.economist.com/the-economist-explains/2018/05/29/why-ubers-self-driving-car-killed-a-pedestrian.

[8] S. S. Banerjee *et al.*, "Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data," in *Proc. 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018.

[9] C. Fan *et al.*, "DryVR: Data-Driven Verification and Compositional Reasoning for Automotive systems," in *Computer Aided Verification*. Springer International Publishing, 2017, pp. 441–461.

[10] E. M. Clarke and O. Grumberg, "The model checking problem for concurrent systems with many similar processes," in *Proc. Temporal Logic in Specification*, 1987, pp. 188–201. [Online]. Available: https://doi.org/10.1007/3-540-51803-7_26

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach," in *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, 1983, pp. 117–126. [Online]. Available: https://doi.org/10.1145/567067.567080

[12] J. R. Bitner *et al.*, "Efficient algorithmic circuit verification using indexed bdds," in *Digest of Papers: 24th Symposium on Fault-Tolerant Computing*, 1994, pp. 266–275. [Online]. Available: https://doi.org/10.1109/FTCS.1994.315633

[13] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," in *Int. Proc. Test Conference*. IEEE, 1998, pp. 990–999.

[14] R. K. Roy *et al.*, "Compaction of atpg-generated test sequences for sequential circuits," in *Digest of Technical Papers, 1988 IEEE International Conference on Computer-Aided Design*, 1988, pp. 382–385. [Online]. Available: https://doi.org/10.1109/ICCAD.1988.122533

[15] I. Hamzaoglu and J. H. Patel, "Deterministic test pattern generation techniques for sequential circuits," in *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design*, 2000, pp. 538–543. [Online]. Available: https://doi.org/10.1109/ICCAD.2000.896528

[16] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, April 1997.

[17] L. Fraade-Blanar *et al.*, "Measuring automated vehicle safety," 2018.

[18] M. T. Review, "many cars have a hundred million lines of code," https://www.technologyreview.com/s/508231/many-cars-have-a-hundred-million-lines-of-code/.

[19] A. Hawkins, "Nvidia says its new supercomputer will enable the highest level of automated driving," The Verge Oct. 10, 2017 https://www.theverge.com/2017/10/10/16449416/nvidia-pegasus-self-driving-car-ai-robotaxi.

[20] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 365–376.

[21] T. Karnik and P. Hazucha, "Characterization of soft errors caused by single event upsets in CMOS processes," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 128–143, 2004.

[22] M. Musuvathi *et al.*, "Finding and reproducing heisenbugs in concurrent programs." in *OSDI*, vol. 8, 2008, pp. 267–280.

[23] S. K. S. Hari *et al.*, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 249–258.

[24] G. Li *et al.*, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proc. International Conf. for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 8:1–8:12.

[25] NHTSA, "Automated Driving Systems: A Vision for Safety," https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13069a-ads2.0_090617_v9a_tag.pdf, 2017.

[26] A. Abdulkhaleq *et al.*, "A Systematic Approach Based on STPA for Developing a Dependable Architecture for Fully Automated Driving Vehicles," *Procedia Engineering*, vol. 179, pp. 41–51, 2017.

[27] N. Leveson, "A new accident model for engineering safer systems," *Safety science*, vol. 42, no. 4, pp. 237–270, 2004.

[28] K. Pei *et al.*, "DeepXplore: Automated whitebox testing of deep learning systems," in *Proc. of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[29] B. Salami, O. Unsal, and A. Cristal, "On the Resilience of RTL NN Accelerators: Fault Characterization and Mitigation," *arXiv preprint arXiv:1806.09679*, 2018.

[30] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 17.

[31] S. Jha *et al.*, "AVFI: Fault Injection for Autonomous Vehicles," in *Proc. 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 55–56.

[32] A. H. M. Rubaiyat, Y. Qin, and H. Alemzadeh, "Experimental resilience assessment of an open-source driving agent," *arXiv preprint arXiv:1807.06172*, 2018.

[33] K. J. Åström and T. Hägglund, *PID controllers: theory, design, and tuning*. Instrument Society of America Research Triangle Park, NC, 1995, vol. 2.

[34] S. M. Erlien, S. Fujita, and J. C. Gerdes, "Safe driving envelopes for shared control of ground vehicles," *IFAC Proceedings Volumes*, vol. 46, no. 21, pp. 831–836, 2013.

[35] S. J. Anderson, S. B. Karumanchi, and K. Iagnemma, "Constraint-based planning and control for safe, semi-autonomous operation of vehicles," in *Proc. 2012 IEEE Intelligent Vehicles Symposium (IV), 2012 IEEE*, pp. 383–388.

[36] A. Avizienis *et al.*, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[37] S. J. Julier and J. K. Uhlmann, "New extension of the Kalman filter to nonlinear systems," in *Signal pfrocessing, sensor fusion, and target recognition VI*, vol. 3068. International Society for Optics and Photonics, 1997, pp. 182–194.

[38] J. Pearl, "Theoretical impediments to machine learning with seven sparks from the causal revolution," 2018.

[39] S. M. LaValle, *Planning algorithms*. Cambridge University Press, 2006.

[40] P. L. DeVries and P. Hamill, "A first course in computational physics," 1995.

[41] D. Koller *et al.*, "Towards robust automatic traffic scene analysis in real-time," in *Pattern Recognition, 1994. Vol. 1-Conference A: Computer Vision & Image Processing., Proceedings of the 12th IAPR International Conference on*, vol. 1. IEEE, 1994, pp. 126–131.

[42] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014.

[43] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society. Series B (methodological)*, pp. 1–38, 1977.

[44] SAE International, *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, Sep 2016.

[45] A. B. Watson and J. A. Solomon, "Model of visual contrast gain control and pattern masking," *JOSA A*, vol. 14, no. 9, pp. 2379–2391, 1997.

[46] P. Debevec and S. Gibson, "A tone mapping algorithm for high contrast images," in *13th Eurographics Workshop on Rendering: Pisa, Italy*. Citeseer, 2002.

[47] D. Menon and G. Calvagno, "Color image demosaicking: An overview," *Signal Processing: Image Communication*, vol. 26, no. 8-9, pp. 518–533, 2011.

[48] J. Redmon *et al.*, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.

[49] D. Reid *et al.*, "An algorithm for tracking multiple targets," *IEEE Transactions on Automatic Control*, vol. 24, no. 6, pp. 843–854, 1979.

[50] A. Houenou *et al.*, "Vehicle trajectory prediction based on motion model and maneuver recognition," in *Intelligent Robots and Systems, Proc. 2013 IEEE/RSJ International Conference*, pp. 4363–4369.

[51] A. Dosovitskiy *et al.*, "CARLA: An open urban driving simulator," in *Proc. of the 1st Annual Conf. on Robot Learning*, 2017, pp. 1–16.

[52] NVIDIA, "NVIDIA Drive Simulation," https://www.nvidia.com/en-us/self-driving-cars/drive-constellation/, Accessed: 2018-09-02.

[53] Nvidia, "Drive Pegasus," https://www.nvidia.com/en-us/self-driving-cars/drive-platform/, Accessed: 2018-09-12.

[54] NEOUSYS, "nuvo-6108gc gpu computing platform | nvidia rtx 2080-gtx 1080ti-titanx," https://www.neousys-tech.com/en/product/application/gpu-computing/nuvo-6108gc-gpu-computing, Accessed: 2018-11-28.

[55] M. Waskom, "seaborn.boxplot," https://seaborn.pydata.org/generated/seaborn.boxplot.html.

[56] J. M. Anderson *et al.*, “Autonomous vehicle technology: A guide for policymakers,” RAND Corp., Tech. Rep. RR-443-2-RC, 2016.

[57] N. Kalra and S. M. Paddock, “Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?” *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[58] S. Jha *et al.*, “Kayotee: A Fault Injection-based System to Assess the Safety and Reliability of Autonomous Vehicles to Faults and Errors,” in *Third IEEE International Workshop on Automotive Reliability & Test*. IEEE, 2018.

[59] Waymo, “On the Road to Fully Self-Driving,” Waymo Safety Report https://assets.documentcloud.org/documents/4107762/Waymo-Safety-Report-2017.pdf, Accessed: 2017-11-27.

[60] P. Koopman and M. Wagner, “Toward a framework for highly automated vehicle safety validation,” SAE Technical Paper, Tech. Rep., 2018.

[61] J. Lu *et al.*, “No need to worry about adversarial examples in object detection in autonomous vehicles,” *arXiv preprint arXiv:1707.03501*, 2017.

[62] K. Pei *et al.*, “Towards practical verification of machine learning: The case of computer vision systems,” *arXiv preprint arXiv:1712.01785*, 2017.

[63] H. Lakkaraju *et al.*, “Identifying unknown unknowns in the open world: Representations and policies for guided exploration.” in *AAAI*, vol. 1, 2017, p. 2.