# AI-DRIVEN METHODS FOR RESILIENCY AND SECURITY ASSESSMENT: THE CASE FOR AUTONOMOUS DRIVING SYSTEM AND HPC STORAGE SYSTEM

BY

SHENGKUN CUI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Research Professor Zbigniew T. Kalbarczyk

## Abstract

Nowadays, computing systems are used extensively in mission-critical exploration, transportation, scientific study, and manufacturing. With the advances in computation technologies, computing systems have become ever more complex. Due to the system's complexity, it is increasingly hard for humans operators to monitor, assess, and manage the system directly. Moreover, traditional model-based or rule-based assessment techniques cannot provide sufficient coverages because of the wide ranges of use cases and failure modes of complex systems. Recently artificial intelligence (AI)-driven methods are used for timely accurate and high-coverage assessments in complex systems because of their ability to learn from data without explicitly modeling the complex systems. This thesis discusses our work on AI-driven assessment methods—RoboTack, DiverseAV, and Kaleidoscope—in the domain of security (RoboTack) and reliability (DiverseAV, Kaleidoscope) assessment in two critical use cases: autonomous driving systems (ADS) and high-performance computing (HPC) storage systems. We show that by using artificial intelligence and machine learning-based techniques, we can perform high-accuracy, high-coverage security or reliability assessments of large-scale, complex systems efficiently in real time.

## Acknowledgments

I want to express my deepest gratitude to my adviser, Professor Zbigniew T. Kalbarczyk, for providing me with his vision and mentorship of my research during my undergraduate and graduate studies. I thank my colleague Saurabh Jha for his guidance and support of the work presented in this thesis.

I am grateful to my mother and father, Xin He and Dong Cui, who always love, encourage, and support me, along with my aunt Yi He, my uncle Zhong Cui, and my cousin Weixi Wu.

I am grateful to James Cyriac, Haotian Chen, Vikram Anjur, Subho Banerjee, and all other members of the DEPEND Research Group for their continuous support of my studies and research.

Lastly, I am thankful for the friendship of Yangge Li, Zonglin Li, and Tianxing Wang, and many others I know in the ECE and CS departments, for their support and encouragement during many hardships.

# Table of Contents

# Chapter 1: Introduction

Computing systems have become pervasive in our daily lives and are used extensively in mission-critical exploration, transportation, scientific study, and manufacturing. For example, embedded systems are used in critical infrastructure to monitor the production process. Real-time systems are used in autonomous vehicles to control the critical movement of the vehicle. High-performance computing systems (HPCs) and large data centers are used to discover novel scientific knowledge from petabytes of data. These systems are required to have high performance—able to compute and complete their tasks on time and with high reliability—and to provide continuous and uninterrupted service despite disruption due to faults or malicious attacks, as any delays in task completion can cost money or even lives [1]. Therefore, one pressing issue is monitoring and managing the system's hardware and software resources to guarantee high performance and high reliability.

Technological advances lead to increases in both hardware and software complexity of modern computing systems. The complexity of hardware increases because of both the increasing number of hardware components [2] and the use of heterogeneous computer fabrics consisting of CPUs, GPUs, FPGAs (field-programmable gate arrays), and proprietary ASIC (application-specific integrated circuit) accelerators [3]–[5]. The software complexity increases because of the increasingly complex programming logics and the use of artificial intelligence methods that involve the usage of heterogeneous computer fabrics. Due to the complexity, criticality, and scales of those systems, it becomes increasingly hard for humans to monitor, assess, and manage the system directly. To address this, people start to use artificial intelligence (AI)-driven methods to monitor, assess, and manage those systems automatically in real time. AI-driven methods have several advantages: 1) learning-from-data, as AI-driven methods can learn system anomaly from data directly, which eliminates the need to explicitly model complex systems, 2) less human-intervention, as AI-driven methods can process a large quantity of data produced by the system, which allows the human operators to focus on more critical tasks, and 3) the use of heterogenous compute hardware which increases the number of failure modes.

This thesis investigates the use of AI-driven methods in the context of two highly important yet very different applications: autonomous vehicle (AV) and high-performance computing (HPC). We are interested in these applications as they both require continuous monitoring, assessing, and managing the system to ensure high performance and resilience, and each application presents unique domain-specific challenges.

Autonomous driving system (ADS) integrates sensors, software, and hardware responsible for autonomous driving in AV. ADS operates in highly dynamic, human-centered, and safety-critical environments, in which critical faults can cause serious accidents[1], [6]. Comprehensively monitoring and assessing the reliability of ADS is inherently hard using traditional methods like formal verification, because of the following reasons: 1) ADS can encounter infinitely many driving scenarios—countless cases needed to be covered for exhaustive assessment, and 2) modern ADS uses black-box machine learning (ML) techniques such as convolutional neural networks (CNNs) for perception, prediction, and planning, making justifying and verifying the decision of ADS a challenging task.

HPC systems used in data centers are large-scale computing systems with tens-of-thousands of computing, storage, and networking components. HPC system is mission-critical in which critical failures can lead to system-wide outage (SWO) that interrupts multiple running jobs. These jobs can be scientific calculations that would take days to finish, or internet services that serve billions of users. As a result, SWO often results in waste-of-energy and incurs heavy financial losses [7]. Comprehensively monitoring and assessing the reliability of HPC is equally challenging due to the following reasons: 1) a large number of interconnected components result in many failure modes, 2) failure at one component can either be masked or propagate through the system and have different manifestation at different levels of the system, 3) a variety of user applications that stress different components of the system can lead to a variety of failure modes, and 4) component failures and resources contention both result in performance degradation at the user level, making diagnosis harder. These three properties of HPC failures make pinpointing the source of the failures and their reasons particularly hard.

AI-driven methods for system assessment address these challenges by intelligently learning the behaviors and hidden states of complex systems from the operation data, and provide more comprehensive and timely coverage for system reliability, safety, and security assessments. Though the coverage might not be 100% due to these systems' complexities, such tools help the system operators catch and prevent catastrophe failures in mission-critical systems like ADSs and HPC systems to mitigate the consequences. We demonstrate the effectiveness of using AI-driven methods for mission-critical system assessment on both ADSs and HPC systems---one for ADS safety and security assessment, one for ADS fault detection, and one for HPC storage system fault detection and localization. In each of these cases, we show that AI-driven methods learn the state-of-the-art ADS and HPC system state from data without explicitly modeling the system and they are generally applicable to other systems, which makes them powerful system assessment tools.

## 1.1 Thesis Contributions

This work presents three AI-driven methods for resiliency, safety, and security assessment in the context of two applications: ADS and HPC systems. We present the overview and contribution of each work in the following paragraphs.

**RoboTack is an ML-driven malware that targets ADS safety.** RoboTack uses a deep-learning model to determine the attacking timing, which greatly increases the attack success rate against state-of-the-art ADS [8], causing emergency braking in 75.2% of the runs and accidents in 52.7% of the runs. RoboTack presents a powerful framework for assessing ADS susceptibility to malicious attacks and discovering ADS vulnerability in critical driving conditions. For example, RoboTack learns the vulnerability window of the sensor fusion module and attacks around that window to cause a fatal accident. All in all, we believe RoboTack can aid the ADS development by mining the ADS vulnerability before hitting the roads.

**DiverseAV is an AI-aided low-cost and high-accuracy approach for ADS hardware and software faults detection.** DiverseAV detects faults that can cause safety issues by exploiting the input data's temporal diversity. DiverseAV duplicates ADS software and runs them in round-robin mode while using data-driven dynamic thresholds condition on speed fault-free simulations for fault detection. DiverseAV can achieve precision and recall, both of 0.87, on detecting permanent hardware faults and precision and recall of 0.99 and 1.0 respectively on detecting software faults, while incurring little compute overheads. Moreover, the DiverseAV framework is universally applicable to different kinds of ADS with minimum alteration to the original hardware and software.

**Kaleidoscope is an AI-driven real-time failure detection and root-cause analysis framework for HPC storage system.** HPC storage system is a vital component of the modern HPC system that directly impacts performance and reliability. Kaleidoscope uses a probabilistic graphical model (PGM) to capture the system's topology and factor functions to represent the relationships between components and uncertainty in failure paths. We use production data that reflects the real-world usage of the system to train the PGM. We deployed Kaleidoscope on NCSA's Blue Waters Supercomputer's Cray Sonexion storage system and achieve 99.3% accuracy in failure localization and 95.8% in root-cause classification, with less than 0.01% runtime overhead.

Overall, these AI-driven methods can learn and identify the failure modes of complex systems by observing the field data without modeling the system explicitly, making resilience and safety assessment in complex systems tractable and achievable in real time. Overall, our methods show great successes in applying AI-driven methods in the ADS and HPC domains, and we believe these methods are the future direction of complex system monitoring and assessment.

## 1.2 Thesis Structure

The remainder of the thesis will be structured as follows. Chapter 2 will introduce modern ADS architecture and underlying issues to be solved. It will discuss representative modern ADS architecture in greater detail and expose common issues found in modern ADS architecture and the usage of black-box algorithms. Chapter 3-4 will present the work on RoboTack and DiverseAV. Chapter 5 will serve as an overview of the HPC storage system using NCSA's Blue Waters as an example and present problems related to monitoring and failure detection in such systems. Chapter 6 will present the work Kaleidoscope. Chapter 7 will discuss the related works in AV and HPC research fields, with a focus on the theme of reliability and security. Chapter 8 will serve as a discussion and conclusion, in which we will conclude this thesis and discuss future research directions.

# Chapter 2: Autonomous Driving System (ADS): Overview

Chapters 2, 3, and 4 describe the works related to the security and reliability assessment of the ADS. ADS controls the critical behavior of the AV so it must operate reliably in every driving environment. Modern ADSs are prone to security attacks due to 1) easy physical and software access to the vehicle, 2) extensive usage of black-box deep learning-based algorithms, and 3) lack of robust detection methods for adversarial attacks. Our work RoboTack highlights these security vulnerabilities of the ADS by developing a smart malware for ADS security assessment. Besides, ADSs are prone to reliability issues due to the usage of heterogeneous compute architectures consisting of CPUs, GPUs, and ASIC for different compute tasks in the ADS. These components are not 100% reliable, and a reliability issue at the critical moment can cause catastrophic problems, while fully duplicating the ADS can be quite costly. Our work DiverseAV proposes a low-cost robust hardware and software fault detection framework that is universally applicable to almost all ADSs to detect and mitigate critical reliability-rated faults.

We will start by describing the details of the architecture of ADS. ADSs are feedback-based control systems that represent the sensing and computational components (both software and hardware) of AVs. An ADS takes in sensor data at each timestep, infers the data and constructs a world model, makes routing decisions, and outputs control commands to the vehicle actuators to drive the vehicle. Modern ADSs are complex integrations of various hardware and software. This chapter describes the software and hardware components of common types of ADS, modular-based and end-to-end-based, in detail, and provides a summary of issues found in current developments of ADS.

## 2.1 ADS Hardware

ADS hardware consists of sensors and compute-hardware used by the ADS software. A commercial ADS often consists of the following types of sensors: camera, LiDAR, radar, global navigation satellite system (GNSS), and inertial measurement unit (IMU) [8], [9]. Cameras provide high-resolution images describing the vehicle's surroundings, which are used for obstacle recognition, lane recognition and sign and traffic light recognition. Multiple cameras are often used to cover as many view points as possible. LiDARs output 3D cloud points of the surroundings with a field of view of 360 degrees, which are used to detect obstacles and infer their distances. Radars output the distance and positions of the obstacle in front and are used for obstacle distance and velocity detection. One advantage of radar is that it is less susceptible to bad weather or bad lighting conditions than LiDAR and cameras, but radar outputs low-resolution images that are not suitable for distinguishing between different types of objects, so radar is often not used for 3D reconstruction of the obstacles. GNSS is used for localizing the vehicle in a global frame, and IMU is used for measuring the vehicle poses (roll, yaw, pitch) and accelerations along the x, y, z-axes.

ADS compute hardware forms a heterogeneous compute architecture with general-purpose CPU, GPU, and ASIC-based compute accelerators, and proprietary accelerators (either integrated or standalone). The GPU or ASIC-based accelerators are massive parallel architectures that are used to accelerate matrix multiplication, convolution, pooling, and element-wise operations that are typically used in a deep neural network (DNN). These compute components are often duplicated to provide additional compute performance and hardware redundancy. For example, NVIDIA AGX Xavier autonomous driving platform has two NVIDIA Xavier SoCs, and each SoC contains an ARM v8 CPU, proprietary deep-learning accelerator, NVIDIA Volta GPU, and proprietary image and video accelerator, with the option of adding two dedicated NVIDIA TU104 based GPU accelerator [3]. Similarly, Tesla's Full Self-Driving Computer [5] uses two SoCs with ARM CPUs for general-purpose computing, and proprietary neural processing units as parallel compute accelerators.

## 2.2 ADS Software: Modular-based ADS

ADS software runs on ADS compute hardware and provides key functionalities for autonomous driving. At each time step, an ADS software is responsible for perception, prediction, localization, planning, and vehicle control.

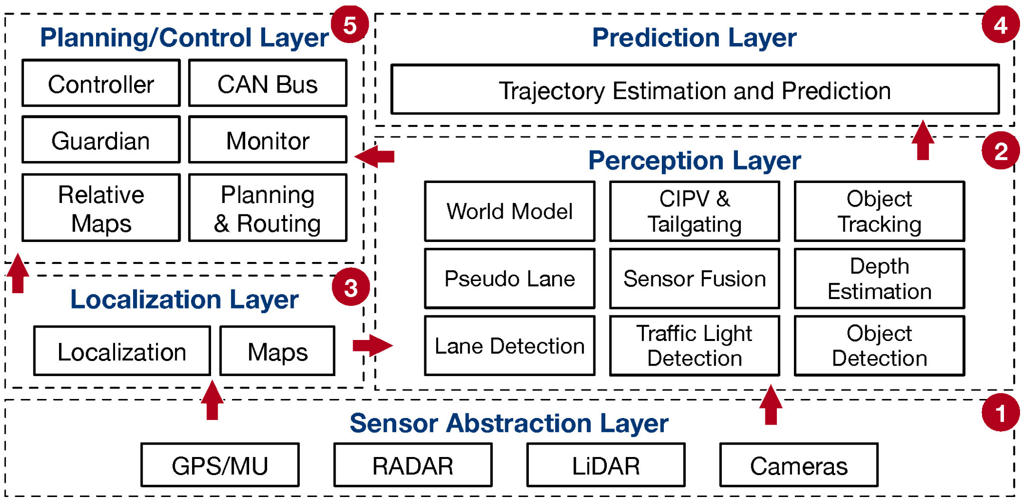


*Figure 1: Modular-Based ADS*
*Source: Adapted from* [33]

Shown in Figure 1, modular-based ADS software such as [8], [9] use individual software modules to solve each of the (from upstream to downstream) perception, prediction, localization, planning, and control tasks. Information is passed from upstream to downstream using a publisher-subscriber paradigm---a module subscribes to topics for input data and publishes topics to broadcast output data. This publisher-subscriber paradigm is often handled by the real-time OS of the ADS. We describe the major components of a modular-based ADS in detail below.

### 2.2.1 Perception Module

Perception is the first step in autonomous driving. The perception module takes the output from cameras, LiDARs, and radars as input, and tracks a list of detected obstacles' positions, velocities, and accelerations over time. This task is called multiple-obstacle-tracking (MOT).

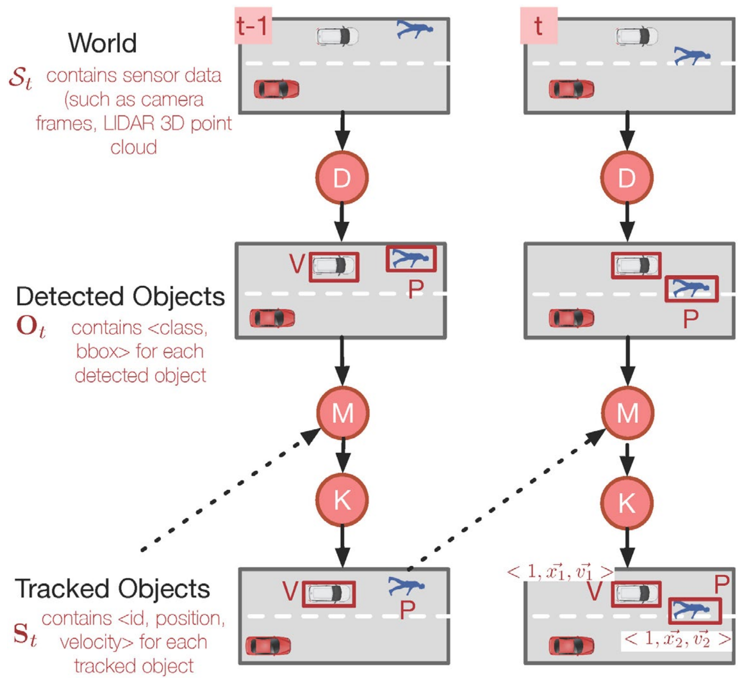

World
$\mathcal{S}_t$ contains sensor data (such as camera frames, LIDAR 3D point cloud
t-1
t
D
D
V
P
P
Detected Objects
$\mathbf{O}_t$ contains <class, bbox> for each detected object
M
M
K
K
Tracked Objects
$\mathbf{S}_t$ contains <id, position, velocity> for each tracked object
V
P
$< 1, \vec{x_1}, \vec{v_1} >$
V
P
$< 1, \vec{x_2}, \vec{v_2} >$

*Figure 2: Tracking-by-Detection Paradigm*

Modular-based ADSs commonly use the tracking-by-detection (TBD) paradigm for solving MOT tasks, for its higher reliability, interpretability, and debuggability compared to black-box end-to-end detection-tracking algorithms. Figure 2 depicts a TBD pipeline for a single sensor such as a camera, a LiDAR, or a radar, and Figure 3 depicts TBD pipelines for multiple sensors with the sensor fusion module.

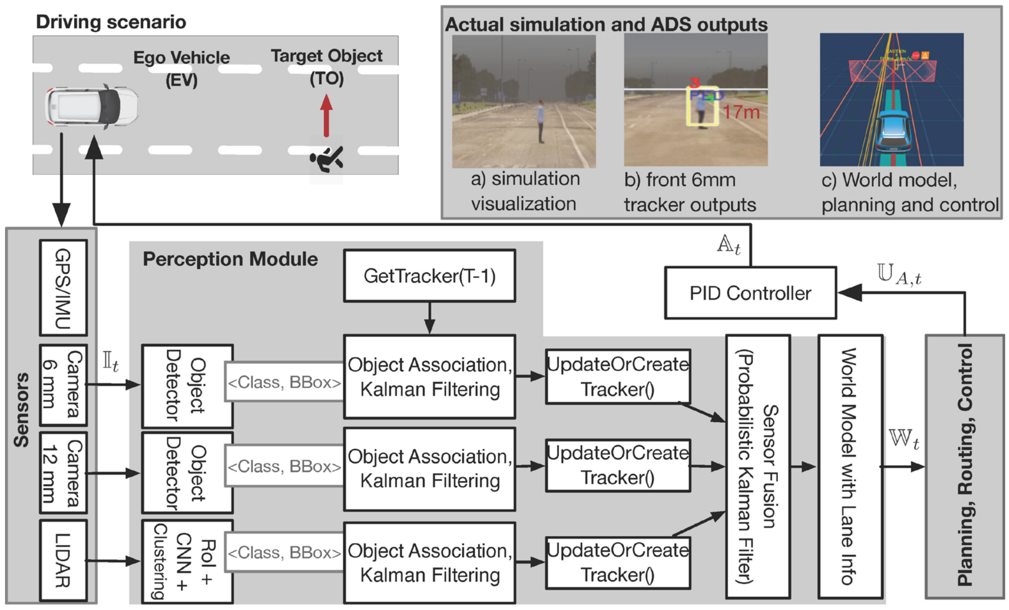


*Figure 3: Multi-sensor Tracking-by-Detection Pipeline with Sensor Fusion Module*

At each sensor time step $t$, a sensor $S_t$, for example, a camera, perceives the environment and generates a sensor output (e.g. camera image). An object detection algorithm $D$ (e.g. a CNN-based object detector) extracts the object states (e.g. position in image), referred to as measurements $O_t$, from the sensor data. Since modern ADS has multiple sensors running asynchronously, there can be multiple instances and types of $D$ for different sensors, each generating an $O_t$ when there is a new output from the corresponding sensor. A matching-algorithm $M$ takes the measurements $O_t$ (possibly from multiple $D$) as well as the existing tracklets $\boldsymbol{S_t}$ (a tracklet is a single tracker instance tracking a single obstacle, multi-object tracking often requires multiple tracklets) and attempts to match the tracklets to their corresponding $O_t$. It is common to transform $O_t$ from different sensors into the same coordinate system (e.g. world coordinate) so that the $O_t$ from different sensors of the same obstacle can all be used to update that obstacle's tracklet. If a $O_t$ is matched with a tracklet, it means that the obstacle measured has already been tracked, so $O_t$ is used to update the tracklet's position by the tracking algorithm $K$. If a $O_t$ is unmatched, it means that there is a new obstacle to track, so a new tracklet is created. If a tracklet is not able to match with any $O_t$, it means that the obstacle is "lost"; the tracking algorithm $K$ can choose to predict the obstacle's position based on its internal motion model, or to delete the tracklet if the obstacle is lost for some time. A way of ranking the correspondence between a tracklet and a measurement is by matching their shape, keypoint features (e.g. SIFT), and comparing their intercept-over-union (IOU is a way to measure area overlaps) with a threshold. The Hungarian algorithm is used to solve this matching-assignment problem given the correspondence of the object features and IOU as criteria. Since an obstacle's tracklet can be updated by different sensor-detection measurements, this process is also called *sensor fusion*. Sensor fusion improves spatial redundancy for obstacle tracking as an obstacle lost by one sensor might still be intact in other sensors. Kalman filter [10] is often used as the tracking algorithm and the sensor fusion model for its ability to track (prediction + estimation update) the obstacle state by using the internal obstacle motion model and updating the obstacle state estimation from different sensor-detector-pipeline's measurements ($O_t$). We omit the details of the Kalman filter as it is beyond the scope of the thesis. The usage of state-estimator models such as KF for tracking solves the following challenges: 1) Sensor outputs are discrete and the obstacles might have moved in intervals. The use of state-estimator can predict obstacle location based on internal motion model, providing priors. 2) Detector outputs are inherently noisy [11], [12] and the state-estimator compensates for the noise by using predicted priors (internal beliefs) and measurement updates. The tracklets $\boldsymbol{S_t}$ are then used with the GPS localization data and the IMU data to construct the world model---a 3D reconstruction of the world specifying the location of the obstacles in the world coordinate.

The perception module, apart from handling obstacle detection, also handles lane detection, sign recognition, and traffic light recognition using camera sensors and additional detection algorithms. Nowadays, specialized CNNs such as LaneNet [13], [14] are used extensively for these tasks.

### 2.2.2 Prediction Module

Prediction is the step after perception. The prediction module predicts the obstacle's motion in the future. Given the obstacle's tracklets' history of $N$ state $\boldsymbol{S_{t-N:t}}$, the prediction module outputs $\boldsymbol{S_{\widehat{t+1:t+T}}}$ the prediction of the tracklets' states until T time steps into the future. These predictions are rough estimations of the obstacle's future trajectory and are used in the planning module for local path-planning. A simple way of trajectory prediction is to use Newton's kinematic model based on trajectory history. This method works fine with vehicles but is less applicable with pedestrians since the latter have more randomness in their movements. There are more advanced techniques such as recurrent neural network-based methods [15], [16].

## 2.2.3 Localization Module

The localization module localizes the ADS position in the regional high-definition maps (HD maps). Different localization techniques are used to localize the ADS to achieve better accuracy. GNSS-based localization localizes the vehicle in the global frame by referring to the localization information sent by orbital satellites. However, a sole GNSS-based localization does not provide the vehicle with heading information or enough accuracy for autonomous driving. Additional methods such as normal distribution transform (NDT) localization matches pre-stored HD map features with the vehicle's sensor measurements (LiDAR measurement in the case of NDT) to determine the vehicle's precise location on the map and its headings. The localization module also provides crucial transform information that transforms both AV and obstacles into the global frame to reconstruct the 3D world representation of the surrounding. Another key component is the HD map. HD map stores detailed information of routes, lanes, and traffic regulations (traffic signs, signals, etc.) used to provide extra information for ADS' path-planning and decision-making.

## 2.2.4 Path Planning Module

With the information from perception, prediction, and planning, the path planning module handles both high-level route planning and low-level motion planning of the AV. The high-level route planner plans the high-level route from current location to the destination. A graph search algorithm such as the Dijkstra algorithm or A* algorithm is used to search the quickest route from the current location to the destination, without considering local obstacles, traffic regulations, or the AV's motion model. A low-level motion planner attempts to plan an obstacle-free path as the AV drives and reaches its destination by searching for a viable path around the obstacles' predicted trajectories, with consideration of the AV's motion capability. Rapidly exploring random tree (RRT) [17] is one efficient algorithm that is commonly used for motion planning. A low-level planning algorithm generates local waypoints for the vehicle's control module to follow.

### 2.2.5 Control Module

Finally, the control module converts the local waypoint from the planning module into control signals for the AV's actuators. Typically, an AV has three main actuators: throttle, brake, and steering, each accepting a range of input. One way of converting local waypoints into smooth control signals (for passengers' comfort) is to use PID controllers [18]. PID controller is a control-feedback loop that computes the control signal based on errors and proportional, integral, and derivative coefficients. The error is calculated between the vehicle's poses (positions and headings) and the next waypoint to track. The control module ultimately outputs throttle, brake, and steering actuation commands to the actuators to drive the vehicle. The use of PID control smooths the output and avoids sudden changes in the control value of the actuator.

## 2.3 ADS Software: End-to-end ADS

Aided by the advancement in deep-learning, end-to-end ADS started to gain much attention [19]–[21] and show promising performance in simple autonomous driving tasks [22]. In contrast to its modular-based counterpart, end-to-end ADS does not have distinctive modules to tackle different tasks of autonomous driving, but learns end-to-end mapping directly from sensor data to planning waypoints, or control output, instead. Many end-to-end ADSs use deep convolutional neural network (CNN)-based models trained by reinforcement or imitation learning to learn such a mapping, and use this mapping directly for autonomous driving. The workflow of an end-to-end ADS is shown in Figure 4.

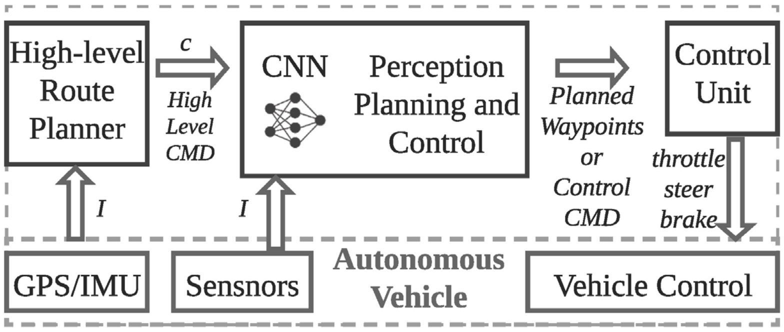


*Figure 4: End-to-End ADS*

Compared to modular-based ADS, end-to-end ADS has its own advantages and disadvantages. One notable advantage is that end-to-end ADS has more flexibility in retaining useful information throughout the perception-prediction-planning-control process [23]. In contrast, a modular-based ADS cannot retain such information because the internal representation of knowledge about the environment in one module is incompatible with another module downstream. For example, perceptual information of the obstacles can be retained throughout the CNN of end-to-end ADS as feature maps and used for planning and control. These feature maps might contain useful information such as position uncertainty and confidence that is critical for more precise planning and control. However, in a modular-based ADS, the output from the perception module is the obstacle's bounding box coordinates instead of the internal feature maps, so the internal representation of the obstacles is lost, which can contain critical information. Another advantage of end-to-end ADS is its simplicity—there is no need to develop and fine tune different modules for different tasks and spend efforts for integration—almost every task is handled by a single model developed and trained end-to-end.

The disadvantages of end-to-end ADS are also prominent. End-to-end ADS often performs worse than modular-based ADS. More importantly, the decision made by end-to-end ADS is less interpretable and verifiable without intermediate outputs of each module like modular-based ADS. For example, when the end-to-end ADS decides to change lanes, we are not able to look at each modules' decision (as with a modular-based ADS) and the reason why the ADS arrived at that decision, but can only suspect that the module learned that from the training data. Moreover, CNN-based end-to-end ADSs are susceptible to adversarial attacks and distribution shifts. The latter matters in particular because out-of-distribution scenarios are common in autonomous driving due to infinitely many possible scenarios. The combination of many possible scenarios and limited interpretability makes it hard to formally verify that the output of an end-to-end ADS is “safe” under distribution shift when encountering new scenarios.

Overall, end-to-end ADS is less commonly used in commercial graded ADS because of its inferior performance and interpretability compared to its modular-based counterpart.

# Chapter 3: RoboTack: ML-Driven Malware Targeting Autonomous Vehicle Safety

This chapter introduces RoboTack [1] [24], a smart ML-based Malware targeting autonomous vehicle safety by attacking the perception module of the autonomous driving system (ADS). RoboTack uses ML to infer scenarios and time for attacking the ADS. Through RoboTack, we have shown that, contrary to common belief, even state-of-the-art ADS with multiple sensors can be highly susceptible to malicious attacks when the ADS is attacked at a critical scenario and time. Also, RoboTack can be used as a universal and powerful framework to assess the security of the ADS by mining the failure patterns in critical driving scenarios.

## 3.1 RoboTack: Overview

Security attacks that violate the safety constraints of AV on road can lead to serious accidents. Creating a security hazard that results in serious safety violations

[1] RoboTack is accepted and published as the paper *ML-Driven Malware that Targets AV Safety* [24] in the *2020 International Conference on Dependable Systems and Networks*. The thesis author is a co-author (second author) of the paper due to his contribution in paper writing, method development, method evaluation, and experiment data analysis. Specifically, the thesis author designed and developed Safety Hijacker, a core component in RoboTack. The thesis author wrote the sections on methodology, experimental setup, evaluation and results, and conclusion. The rest of the work is mainly credited to Saurabh Jha, the first author of the paper. © 2020 IEEE. Reprinted with permission.

(emergency brakings or accidents) is not only attractive from the perspectives of malicious entities but also helps expose the underlying security problems of an ADS during its development. The security hazard could be created using smart malware that modifies the sensor data at an opportune time to interfere with the ADS' perception and prediction processes during autonomous driving. Such malware makes the ADS miscalculate the trajectories of obstacles, leading to unsafe driving decisions. For example, the a malware can fool the ADS into believing that a slowing-down vehicle in front is in another lane, making the AV accelerate, and causing rear-end collision with the vehicle in front.

In this work, we introduce RoboTack, an end-to-end attack framework that can be deployed in the form of smart malware to compromise the safety of the AV with a high success rate, and evaluate its effectiveness on a production-grade ADS. The key contributions of RoboTack are that it answers the questions of “what”, “how”, and “when” to attack the ADS and it proposes a general framework for ADS security assessment.

Deciding what to attack: RoboTack selects what obstacle to be attacked as well as what attack method should be used to attack. RoboTack has situation awareness of its surroundings so it selects the obstacle and attack vector to maximize the impact of the attack using a rule-based system.

Deciding how to attack: RoboTack smartly alters the obstacle's predicted trajectory by altering the camera sensor input data or output of the perception module and maintaining that trajectory for a short time interval. RoboTack decides the attack method and attack duration in answering "what to attack" such that the alteration is small enough to be considered as noise. RoboTack also overcomes compensation from other sensors (e.g. LiDAR) and temporal stat-estimation models (e.g. Kalman filter).

Deciding when to attack: RotoTack employs a scenario matching algorithm and a shallow three-layer neural network (NN) decision model to identify the most opportune time to cause a safety hazard with a high probability of success. The proposed NN models the non-linear relationship between AV kinematics and attack parameters (how long to attack) and is used for attack decision-making. We use multi-layer NN because it is a universal function approximator [25].

We deploy RoboTack on Apollo ADS [8], a modular-based production-grade AV system to evaluate and quantify the effectiveness of RoboTack, by simulating about 2000 runs of experiments with realistic traffic configuration using LGSVL simulator [26]. We find that 1) RoboTack is significantly more successful in creating safety hazards than random attacks (our baseline) are. Random attacks attack random obstacles at random moments and for a random duration, without intelligent decision-making. RoboTack caused 33x more forced emergency braking than random attacks. 2) RoboTack causes accidents in 52.6% of the experiments, while random attacks caused none. 3) RoboTack is more successful in causing an accident when a pedestrian is involved, causing accidents in 84.1% of the runs. 4) RoboTack automatically learns the vulnerability of Apollo's perception system from data and can achieve successful attacks in as few as 14 consecutive time steps (less than a second).

## 3.2 RoboTack: Related Work

We compare RoboTack with other adversarial learning methods targeting perception systems or AV safety. Several works have targeted deep neural networks (DNNs) to create adversarial learning-based attacks [27]–[31] that were shown to be successful in causing misrecognition of the DNNs. Moreover, [32] proposed a method to overcome the temporal redundancy of the Kalman filter which leads to mistracking of the obstacles. The theme of these researches is to create adversarial input or obstacles on-road in an attempt to "fool" the AV's perception system. However, these adversaries are limited because 1) DNNs only represents a part of the system, 2) have low safety impact because tracking and sensor fusion provide temporal and spatial redundancy to protect the vehicle from adversarial attacks while existing methods only consider a single sensor pipeline, and 3) existing methods lack attack timing and scenario awareness, so most attacks are carried out at random which will not cause safety violations of AV. Overall, existing adversarial attack methods only focus on "what and how to attack" instead of "what, how, and when to attack" like RoboTack to improve attack success rate. Moreover, existing adversarial methods do not consider the end-to-end autonomous driving feedback loop for evaluation but only evaluate precaptured data. In contrast, we evaluate RoboTack on a production-grade AV system with an end-to-end ADS-simulator feedback loop for a more realistic evaluation.

## 3.3 RoboTack: Background

1) Modular-based ADS: Figure 1 illustrates the basic architecture of modular-based ADS. From now on, we use AV, and ego vehicle (EV) interchangeably. A detailed overview of modular-based ADS is presented in Chapter 2 so we omit the details here. In short, at each time step $t$, the ADS system takes input from sensors $\boldsymbol{I_t}$ (e.g. cameras, LiDAR, GPS, IMU) and reconstructs a world model $\boldsymbol{W_t}$ consisting of position and velocity of surrounding obstacles, and lane and path information. The planning module uses $\boldsymbol{W_t}$ to plan a viable path to the destination. The path information is converted into steering, throttle, and brake actuation values $\boldsymbol{A_t}$ using PID controllers.

2) Perception system: Chapter 2 presents the perception system and multiple-obstacle-tracking (MOT) paradigm in-depth so we omit the details here. As shown in Figure 3, the sensor input $\boldsymbol{I_t}$ is the input to a CNN-based object detector, such as YOLO [11] or Faster R-CNN [12]. The obstacle detector extracts a list of obstacles from $\boldsymbol{I_t}$ known as measurements; we named it as $\boldsymbol{O_t}$. The obstacle tracker, like Kalman filter (each obstacle has one), tracks the obstacle over time, by first generating state prediction $\widehat{\boldsymbol{S_{t-1}}}$ and generating state estimation $\widehat{\boldsymbol{S}}_{\boldsymbol{t}}$ based on measurements. $\widehat{\boldsymbol{S}}_{\boldsymbol{t}}$ is then used with the appropriate coordinate transform T to build the world model $\boldsymbol{W_t}$.

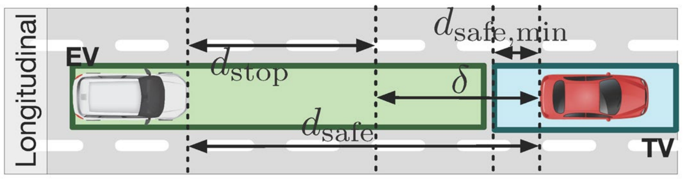


*Figure 5: Autonomous Vehicle Safety Model*

3) Safety model: A safety model is a model to evaluate the safety of the autonomous vehicle quantitively. We use the safety model proposed by [33] as it defines the safety criteria of an AV in terms of longitudinal and lateral distances from the closest-in-path-obstacle (see Figure 5). Since in this work all our scenarios are driving in a straight-line, we use only the longitudinal definition. The stopping distance $d_{stop}$ is the distance needed to bring AV to stop under maximum deceleration. The safety potential $\delta$ is the distance from $d_{stop}$ to the nearest obstacle in the path. Finally $d_{safe} = d_{stop} + \delta$, and the AV is collision-free as long as $d_{safe} > 0$. A larger safety potential requires a shorter stopping distance or keeping a longer distance with in-path obstacles. On the other hand, reducing safety potential is equivalent to decreasing $d_{safe}$ given the same stopping distance. In this work, we use $d_{safe} > 4$ because a limitation in the simulation environment terminates the simulation for $d_{safe} < 4$ m.

## 3.4 RoboTack: Attack & Threat Model

1) Attacker's goals: The attacker ultimately aims to alter (hijack) the obstacle's trajectories to cause a safety hazard. Safety hazards in this work are emergency brakings and accidents. The attacker needs to stay hidden by short attack duration and

disguising the attack as natural noise in the perception pipeline. RoboTack achieves this by hiding the data perturbation of the sensor data as sensor noise and the resulting perturbed output of the detector as detector noise (discussed in Section 3.7). Taking advantage of the natural noise of the perception pipeline the malware attacks at a critical time—causing the maximum impact on safety potential for a minimum duration. This makes it hard for the intrusion-detection system (IDS) to detect [34] in time. Attacking at the critical time requires the malware to be scenario-aware of AV's surroundings as altering the obstacle trajectories alone is not sufficient to cause an attack if there is no obstacle nearby. The malware can automate the process by continuously monitoring the surroundings, identifying the critical time, and attempting to cause a safety hazard.

2) Threat model: Here we discuss the thread model, including system defensibility and attacker's capability. The targeted system is the perception subsystem of the ADS, specifically the obstacles detection, tracking, and sensor fusion modules. The built-in defenses are the spatial redundancy provided by multiple sensors and the sensor fusion and the temporal redundancy provided by the obstacle state-estimator (tracker). These two built-in redundancies defend against most of the existing adversarial attacks on detectors [35] since these attacks focus on only a single sensor and attack at a random time. The malware targets the critical vulnerable component in the perception module—the Kalman filter (KF as $K$ in Figure 2) used as obstacle trackers.

KF assumes Gaussian noise with zero mean for both the measurement and the process update, which is true for the bounding box predicted by the CNN-based detectors. However, these assumptions make KF less effective in compensating for adversarial noise (added to measurements) that is tailored to alter the obstacles' trajectories because KF cannot identify noise that violates the assumptions. We show later that the malware can alter the trajectory of a perceived obstacle within one standard deviation of the modeled Gaussian noise and result in successful attacks. Since a KF uses an obstacle's trajectory history in trajectory estimation (temporal redundancy), one challenge of attacking KF is to maintain a small window (continuous-time step of the attack) to avoid IDS detection, while still resulting in an effective attack (too small a window will not alter the trajectory by much).

We assume the attacker can get access to the ADS source code and live camera feed, or get access to the ADS perception module during its operation. The attacker first gains knowledge of the ADS system and understands its internal components. The code can be obtained via a rogue employee. Moreover, there are works such as [36] that showed the steps to intrude the ethernet-based camera and spoof the camera traffic using the man-in-the-middle (MITM) attack. This way, the adversary can gain access to the camera data and modify them at a given time. Finally, the attacker can choose to use a hardware implant to compromise the ADS. One way is to install backdoors in the hardware for remote access.

We assume that the attacker cannot get access to the CAN bus and we assume the transmission of control commands is encrypted and protected [37], so the attack cannot perturb the control command of the ADS directly to cause safety hazards. These are reasonable assumptions because the CAN bus is protected on production graded systems. In addition, IDS can detect abnormal activities on the CAN bus.

3) Attack Vectors: Attack vectors are attack configurations to maximize impact depending on the scenarios. The available attack vectors in RoboTack are Move_Out, Move_In, and Disappear, but other attack vectors are also possible. Move_Out attack moves the targeted vehicle (TO) out of the AV lane to fool the ADS into believing that TO is moving out-of-lane and the path at the front is obstacle-free. This attack makes the ADS accelerate and causes it to collide with the TO. Move_In attack moves an out-of-lane TO into the AV's lane, forcing the ADS to perform emergency braking. Emergency braking can cause serious injuries. Disappear attack fools the ADS into believing that in-path TO has disappeared; it is a more extreme version of the Move_Out attack with similar consequences.

4) Attack phases: The first phase of the attack is preparing and deploying the malware, which includes gaining access to ADS source code, defining a mapping between attack vectors and world state ($\boldsymbol{W_t}$), training "safety-hijacker" and optionally fine-tuning "trajectory-hijacker" for a given ADS, gaining access to ADS camera feeds or detector outputs, and installing RoboTack on the targeted ADS.

The second phase of the attack is monitoring the environment. RoboTack first approximately reconstructs the world state using a subset of the sensors (e.g. single camera) to gain scenario awareness. We assume that the world state approximated by the front-facing camera is close to the one constructed using sensor-fusion, but less accurate. The second step is identifying the victim or TO; we greedily choose the closest-in-path-obstacle (CIPO) identified by our safety model introduced in the previous chapter. The third step is invoking scenario matcher (SM) that uses the world state to determine whether the TO is vulnerable to any attack vector. The fourth step is using the "safety hijacker" (SH) to decide the attack timing and duration. The SH decides the timings of the attack by estimating the impact of the attack on the safety potential $\delta$ by using an NN with three hidden layers. SH only launches the attack when $\delta$ is reduced below a predefined threshold of $\gamma$ meters. The malware ensures that the duration of the attack does not exceed some predefined value to avoid IDS detection; this value is determined from the characteristic of obstacle detectors on continuous misdetection in normal (attack-freed) autonomous driving scenarios.

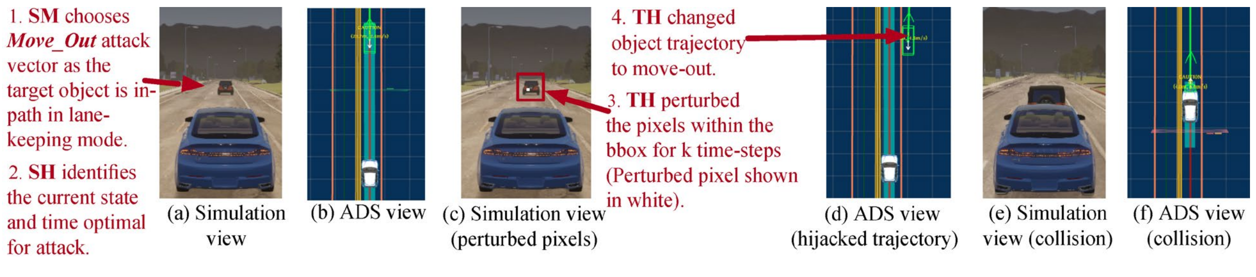

1. SM chooses Move_Out attack vector as the target object is in-path in lane-keeping mode.
2. SH identifies the current state and time optimal for attack.
(a) Simulation view
(b) ADS view
(c) Simulation view (perturbed pixels)
3. TH perturbed the pixels within the bbox for k time-steps (Perturbed pixel shown in white).
4. TH changed object trajectory to move-out.
(d) ADS view (hijacked trajectory)
(e) Simulation view (collision)
(f) ADS view (collision)

*Figure 6: RoboTack Attack Example*

The third phase of the attack is altering the TO's trajectory according to the chosen attack vectors, at the time and for the duration that SH decided. One way to achieve this is to use "trajectory hijacker" (TH) to corrupt the camera feed. TH perturbs the camera sensor data using adversarial patterns to fool the detector so that the obstacle's trajectories are altered to match the attack vector. The TH then attacks TO for the duration chosen by the SH. Another way to achieve this is to directly perturb the bounding box coordinates predicted by the obstacle detector, assuming the attack has access to the detector outputs.

5) Real attack example: In Figure 6 we show an example of using RoboTack to attack Apollo ADS, running with LGSVL simulator. We show two different views: i) a simulation view showing the simulated scenario, and ii) an ADS view showing how the ADS perceives the world. RoboTack continuously monitors every camera frame using the scenario matcher (SM) to identify the targeted obstacle (TO) and attack vectors. If SM does not identify such an obstacle, it aborts the attack and retries the next frame. Otherwise, SM invokes the safety hijacker (SH). As shown in the figure, the SM identifies the SUV in front as a TO, and based on the situation, SM selects the "Move_Out" attack vector. SH, upon receiving the signal from SM, starts to infer the reduction in safety potential due to the attack and determines the number attack duration. When the predicted reduced safety potential is reduced below the 10 m threshold the SH launches the attack by calling the trajectory hijacking (TH) to change

the SUV's trajectory. TH changes the trajectory of the SUV by perturbing the camera feed with a small patch on the SUV (the white patch for illustration purposes). The resulting trajectory perceived by the ADS is shown in the fourth plot in Figure 6, in which the SUV is moved out of the lane. ADS decides to accelerate because of this and collides into the SUV as shown in both the simulation view and the ADS view.

## 3.5 RoboTack: Methodology

In this section, we discuss the core components of RoboTack—scenario matcher (SM), safety hijacker (SH), and trajectory hijacker (TH)—in detail.

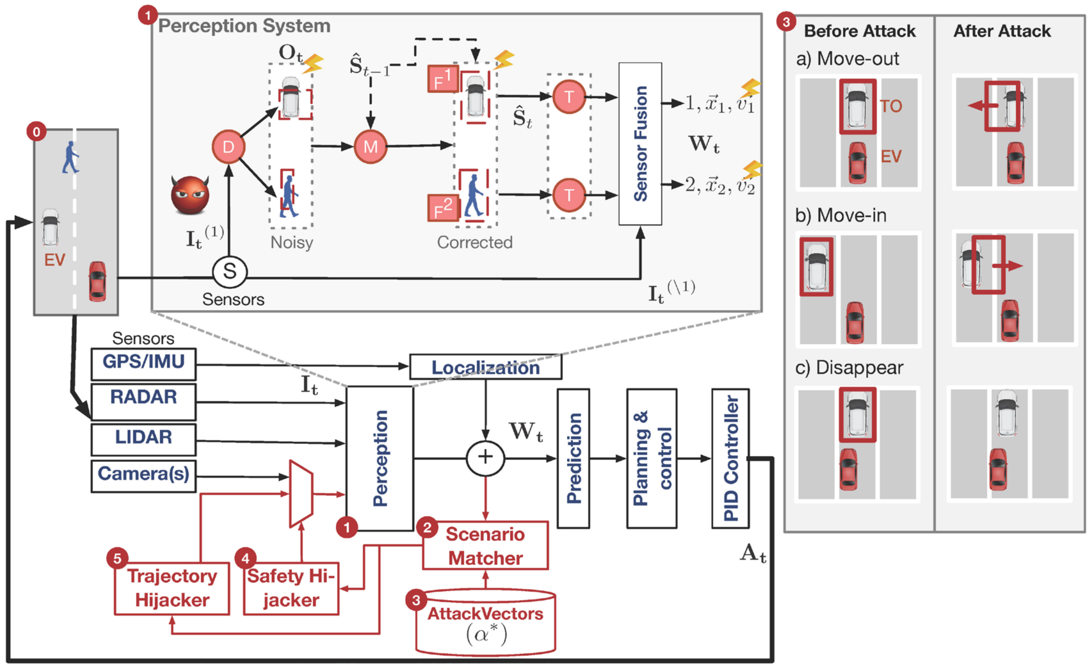


*Figure 7: RoboTack ADS and Attack Overview*

We begin by summarizing the entire attack procedure into three key steps: 1) monitoring the environment and selecting the candidate attack vector and targeted obstacle using SM, 2) deciding when to attack and attack duration based on the selected attack vector and targeted obstacle using SH, and 3) launching the attack by perturbing the trajectory of the targeted vehicle using TH. An overview of the attack is summarized in Figure 7.

*Table 1: Scenario Matching Attack Vector, TO: Target obstacle*

| **TO Trajectory** | **TO In AV-lane** | **TO not in AV-lane** |
|---|---|---|
| Moving In | --- | Move_Out/Disappear |
| Keep Lane | Move_Out/Disappear | Move_In |
| Moving Out | Move_In | --- |

1) Scenario matcher: The goal of the SM is to monitor the environment and check whether the targeted obstacle is vulnerable to any of the attack vectors. We choose greedily the closest-in-path-obstacle (CIPO) to be the targeted obstacle (TO). SM is essential to maximize the success rate and minimize chances of IDS detection because the SM avoids launching the attack if there is no CIPO in front that would affect the ADS, or if the CIPO is performing a maneuver similar to that of the attack vector (moving-out when the attack vector is Move_Out). SM provides RoboTack with scenario awareness. SM uses a rule-based system to select the attack vector, and the rules are summarized in Table 1. Note that SM can interchangeably choose between Move_Out and Disappear attack vector as they have the same consequences on the target's trajectory. However, we found that Disappear which requires a more sudden perturbation in trajectory is better suited for targeting pedestrians because the attack duration is small. In contrast, the attack duration required for the vehicle is large so

Move_Out has a better chance to "stay under the radar" of the IDS due to a more graduate change.

2) Safety hijacker: The SH module decides when to attack to maximally impact the vehicle's safety. The SH optimally attacks the vehicle when the attack results in a safety potential of fewer than 4 m. The SH runs at each time step to determine the attacking timing and attack durations. At each time step $t$, the SH takes the difference between the AV and TO velocities and accelerations, the safety potential $\delta_t$ to the TO, and the attack vector $\alpha$ as inputs, and outputs the attack decisions. SH outputs the attack duration $K$ specifying the number of consecutive camera frames to attack if the decision is to attack the TO. However, predicting $K$ directly is hard because it is unbounded and sparse. We therefore construct the learning problem as follows: given the relative velocity and acceleration and the safety potential of TO, what is the predicted safety potential after $K$ consecutive frame attacks with the selected attack vector? Here the learning problem becomes predicting safety potential under the attacks, which can be directly collected from simulation data. $K$ and $\alpha$ become a tweakable hyperparameter in the new formulation. In other words, we need to learn a function $f_\alpha$ such that $\delta_{t+k} = f_\alpha(v_{rel}, \alpha_{rel}, \delta_t, k)$. The SH only issues the attack command when $\delta_{t+k} < \gamma$, and we use $\gamma = 4$ for all of our experiments, which is the distance of a "crash" in the simulator. $\alpha$ is selected by the SM before invoking the SH. To obtain the optimal $k$ which we refer to as $K$, RoboTack uses a parameter-sweeping strategy on $k$ to

search for the optimal value $K = argmin_k\, k \cdot (\mathbf{I}[\delta_{t+k} \leq \gamma])$, subject to $k \leq K_{max}$, with I the indicator function. We use the fact that for all of our scenarios, $f_\alpha$ is non-increasing with increasing $k$ (also shown by our data), so that the first $k$ that satisfies the condition is the optimal point. We can also use binary search.

SH estimates $f_\alpha$ using a three-layer feed-forward NN. We choose NN because it is a universal function approximator [25] that helps to model the uncertainty in the ADS. Note that the malware uses a uniquely trained NN for each attack vector. The input to the NN is the vector $[v_{rel}, \alpha_{rel}, \delta_t, k]$ and the output of the NN is $\delta_{t+k}$, the safety potential after $k$ consecutive frames of attack, given the input and the attack vector. Intuitively, NN learns the effect of an attack vector and attack duration on the ADS safety potential. We train the NN using the average L2 distance between the predicted $\delta_{t+k}$ and the groundtruth $\delta^G_{t+k}$ from the training dataset $\mathcal{D}_{train}$. The NN we used has three hidden layers with 100, 100, and 50 neurons respectively, using the ReLU activation function and dropout layers between hidden layers with a dropout rate of 0.1. The rather simple architecture of the NN is chosen to reduce the inference time with sufficient learning power, as we find that further increasing of the number of hidden layers results in limited improvement. The NN's prediction error is within 1 m for pedestrians and within 5 m for vehicles.

We trained the NN using a subset of $\mathcal{D}$ collected from a series of driving simulations for different scenarios, with Baidu's Apollo ADS and LGSVL Simulator. To collect training data we randomly and uniformly choose $\delta_{inject}$ for each run from a range between 70 m (the start of the scenario) and 4 m, and a random $k$ between 10 and 100. The data collector invokes the trajectory hijacker (TH) as soon as the safety potential reaches $\delta_{inject}$. The NN is optimized with Adam optimizer and trained on a 60%--40% split of the dataset $\mathcal{D}$ for training and validation.

3) Trajectory hijacker: The TH module alters the trajectory of the CIPO according to an attack pattern, by either perturbing the camera feed or the output of the obstacle detector (obstacle bounding boxes). The TH must also hold onto the attack for $k$ consecutive frames to mitigate the temporal redundancy from the Kalman filter.

The objective of TH when used with the camera feed is to perturb the camera feed in such a way that the bounding box $\widehat{s_t^i}$ estimated by the MOT algorithm at time $t$ moves in a given direction (left or right) at max by $\omega_{max}$. This objective can be formulated as an optimization problem, and to solve it we modified the formulation provided by Jia et al. [32]. The core idea is to find the adversarial patch corresponding to a translation vector $\widehat{w_t}$ at each time step that results in the maximal Hungarian matching cost with tracker evasion (Disappear) or without tracker evasion (Move_Out): $max_{\widehat{w_t}}\, M\left(o_t^i + \widehat{w_t},\ \widehat{s_{t-1}^i}\right)$, where $o_t^i$ is the obstacle detector's output, $\widehat{s_{t-1}^i}$ is the prediction of the obstacle bounding box by the internal motion model, and $o_t^i + \widehat{w_t} = Detector(I\ +\ patch)$, meaning that applying an adversarial patch on the input of the detector results in the shifting in position of the bounding box. Additional constraints need to be satisfied: 1) $M \leq \lambda$, meaning that the perturbed bounding box cannot escape the original tracker unless the attack vector is Disappear, 2) $\widehat{w_t} \in [\mu - \sigma, \mu + \sigma]$, meaning that the movement must be disguised as noise, and 3) $IOU\left(o_t^i + \widehat{w_t}, patch\right) \geq \gamma$, meaning that the adversarial patch must be applied on the CIPO. It is also important to stop maximizing the distance between the actual obstacle state and the perturbed obstacle state once $\Omega_{max}$ and hold on to that to maintain the object's trajectory. We omit the details of optimization. This process is simplified if one can directly access and alter the output of the obstacle bounding boxes (used in our evaluations). For example, with direct access to the output, there is no need to search the adversarial patch; rather, directly apply a vector $\widehat{w_t}$ to translate the bounding box with the desired amount.

4) Implementation: We implemented RoboTack using Python for the attacker and C++ for the ADS integration. We assume that we can access the output of the obstacle detector directly for TH. The proposed malware (RoboTack) has a small footprint of fewer than 500 lines of code and a 4% additional GPU utilization with negligible additional CPU utilization. It is therefore difficult to detect the usage of RoboTack by monitoring the system's resource utilization.

## 3.6 RoboTack: Experimental Setup

1) ADS: We use Baidu's Apollo [8] ADS for autonomous driving and LGSVL Simulator [26] for simulation. Apollo ADS uses multiple sensors: a LiDAR and two

front-facing cameras for perception, GPS for localization, and IMU for pose estimation. We enable all sensors for our experiments.

2) Simulator: We use the LGSVL simulator to simulate driving scenarios. LGSVL simulator is a realistic driving simulator that simulates the driving environment, obstacles, physical interactions, and virtual sensor data. A driving scenario is a predefined environment, with predefined actors and a series of events. For example, at the 5th second the NPC car in front will change lanes. LGSVL posts virtual LiDAR, camera, IMU, and GPS data to the ADS as if the ADS is driving in the real world. The measurements of different sensors are posted in different frequencies: cameras at 15 Hz, LiDAR at 10 Hz, and GPS and IMU at 12.5 Hz. At the time of this work, LGSVL does not support radar sensors with Apollo. Finally, LGSVL provides Python API for scenario creation.

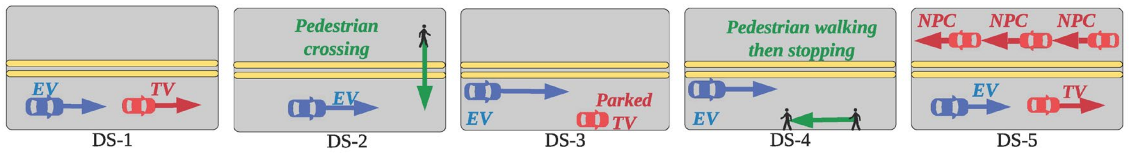


*Figure 8: Driving Scenarios*

3) Driving scenarios: We use LGSVL's Python API to create five common and high-risk driving scenarios. All scenarios are based on the "Borregeas Avenue" located in Sunnyvale California with a speed limit of 50 kph, unless otherwise specific, the AV cruise at 45 kph without frontal obstacles. We depict all driving scenarios in Figure 8.

In driving scenario 1 or DS-1, the ego vehicle (EV, or AV) followed a targeted vehicle (TV or NPC) traveling at a constant speed of 25 kph. The TV started 60 m ahead of the AV. In the golden (attack-freed) run, the AV should accelerate to 45 kph while approaching the TV, and then slow down to 25 kph to match the speed of the TV while maintaining a 20 m distance. We use this scenario to evaluate Disappear and Move_Out attacks on a vehicle.

In driving scenario 2 or DS-2, a pedestrian illegally crossed the street about 40 m away from the AV. In the golden run, ADS commands the AV to brake and avoid collision with the pedestrian and stop at 10 m in front of the pedestrian. The AV starts moving again once the pedestrian moves off-road. We use this scenario to evaluate Disappear and Move_Out attacks on a pedestrian.

In driving scenario 3 or DS-3, a targeted vehicle is parked on the side of the road in the parking lane. The AV should maintain its trajectory because the parked vehicle is not an in-path obstacle. We use this scenario to evaluate the Move_In attack on a vehicle.

In driving scenario 4 or DS-4, a pedestrian walks longitudinally towards the AV in the parking lane (next to the AV's lane). The AV should reduce speed to around 35 kph while passing the pedestrian, and accelerate back to 45 kph once it passes the pedestrian. We use this scenario to evaluate the Move_In attack vector on a pedestrian.

In driving scenario 5 or DS-5, there are multiple vehicles traveling both in the AV's lane and on the opposite lane, with random destinations. Also, there are randomly generated pedestrians on the walking-lane. The AV is set to follow a target vehicle just like DS-1, but this time with multiple non-AV vehicles in addition to just the one in front. We use this scenario as the baseline for random attacks for comparison.

4) Hardware platform: The production version of the Apollo ADS is supported on commercially available hardware like the Nuvo-6108GC [38], which consists of Intel Xeon CPUs and NVIDIA GPUs. We therefore run the ADS on a workstation with a Xeon CPU, ECC memory, and NVIDIA Titan XP GPUs, with one dedicated GPU running the simulator and the other running the ADS.

## 3.7 RoboTack: Evaluations and Discussion

1) Characterizing ADS perception system on obstacle detection: We characterize the performance of Apollo's perception system which uses YOLO [11] for obstacle detection. We measure 1) the distribution of consecutive frames in which an obstacle is continuously undetected, and 2) the distribution of the error in the center positions of predicted bounding boxes compare to the groundtruth. The goal of this characterization is to show that RoboTack attacks are indistinguishable from detector noise. We show that RoboTack's attacks fall in the 99th percentile of the corresponding distribution in both aspects, and neither can be detected by IDS. For such characterizations we use

sequences of recorded camera frames and the groundtruth labels by manually driving the vehicle on the San Francisco map for 10 min with regular traffic.

Continuous misdetection: An obstacle is misdetected if the IoU between the predicted bounding box and the groundtruth is less than 60%. We quantify pedestrian detection and vehicle detection separately. Both cases follow an exponential distribution, with the $99^{\text{th}}$ percentile of continuous misdetection being 31 frames for a pedestrian and 59 frames for vehicles.

Bounding box prediction error: The bounding box prediction error is calculated as the difference between the predicted bounding box by Apollo's MOT tracking pipeline (YOLOv3 with Kalman filter) and the simulator groundtruth. We normalize the differences by the size of the groundtruth bounding boxes and we evaluate pedestrian detection and vehicle detection separately. The distribution follows a Gaussian noise distribution with a $99^{\text{th}}$ percentile of 1 object length for both pedestrian and vehicle detections.

*Table 2: Attack Result Summary. DS-5 baseline random attack in bold. EB: emergency braking without accidents, Accidents: crashes into obstacles, R: RoboTack Full. $\widehat{K}$ in baseline attack is picked between 15 to 85 for each run of experiment.*

| **Exp ID** | **K** | **# runs** | **# EB (%)** | **# Accidents (%)** |
|---|---|---|---|---|
| DS-1-Disappear-R | 48 | 101 | 54 (53.5%) | 32 (31.7%) |
| DS-2-Disappear-R | 14 | 144 | 136 (94.4%) | 119 (82.6%) |
| DS-1-Move_Out-R | 65 | 185 | 69 (37.3%) | 32 (17.3%) |
| DS-2-Move_Out-R | 32 | 138 | 135 (97.8%) | 116 (84.1%) |
| DS-3-Move_In-R | 48 | 148 | 140 (94.6%) | --- |
| DS-4-Move_In-R | 24 | 135 | 106 (78.5%) | --- |
| **DS-5-Baseline-Random** | $\widehat{K}$ | 131 | 3 (2.3%) | 0 |

2) Quantifying baseline attack success rate: We use DS-5 as our driving scenario for baseline random attacks. Random here means we (i) choose a random targeted obstacle (TO), (ii) choose a random attack vector, (iii) initiate the attack at a random moment, and (iv) continue the attack for a random duration (randomly chose $K$). Baseline random attack uses neither the scenario matcher (SM) nor the safety hijacker (SH), but only uses the trajectory hijacker (TH) to modify the trajectory of the TO. Out of the 131 valid experiments of random attacks using DS-5, the AV performs emergency braking in only 3 runs (2.3%) and crashes 0 times (see Table 2 DS-5).

We also compare the full RoboTack malware with RoboTack with the SH disabled that only uses SM and TH. These attacks are randomly initiated for random durations $K$ and we evaluate these attacks using DS-1 to DS-4 as a direct comparison with the full RoboTack malware (SM+SH+TH). Through these attacks, we show that attack timing is critically important to successful attacks. We summarize these attacks in Table 2 and the green box plots of Figure 9, with details described in a later section. Together, the completely random attacks and attacks with SM and TH only are analogous to the attack formulations in [27], [28], [31] respectively.

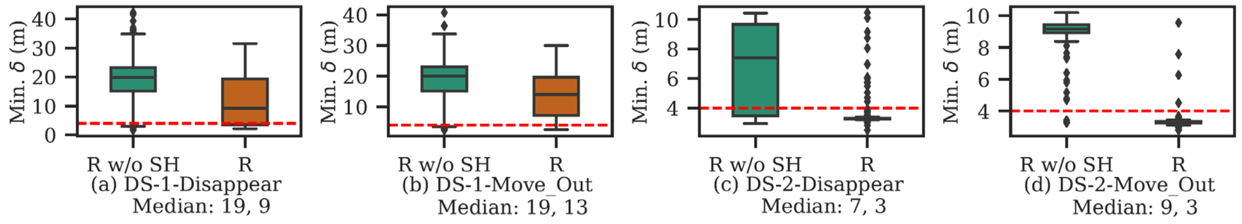


*Figure 9: Impact of Attacks. $\delta$: Safety potential; R w/o SH: RoboTack without safety hijacker; R: Full RoboTack; Red dash line: safety potential $\delta = 4$*

3) Quantifying RoboTack attack success rate: The full RoboTack system uses SM, SH, and TH to complete the attack procedure. Table 2 summarizes the average value of the attack duration $K$, experiment runs, number of emergency brakings, and number of crashes for each attack scenario and attack vector. Each ID in Table 2 specifies the scenario and the attack vector chosen, with 150 to 200 total experiment runs, though we remove invalid experiments (simulator crash, etc.). RoboTack attacks are much more successful in causing safety hazards than random attacks are. RoboTack attack cause 33x more emergency braking (640 out of 851, 75.2%) than random attack (3 out of 131, 2.3%). Random attacks cause 0 accidents while RoboTack attacks cause accidents in 299 out of the 568 runs (excluding Move_In attacks because there is no vehicle to crash into).

RoboTack is more successful in attacking pedestrians than vehicles. We observe that DS-2 and DS-4 have higher safety hazard rates than DS-1 and DS-3. In DS-2 and DS-4 experiments, 84.1% of the runs have accidents involving pedestrians, and 87.8% of the runs have emergency braking for Move_Out attacks, and these numbers change to

82.6% and 94.4% for Disappear attacks. For Move_In attacks, there is no accident, but for 78.5% of the runs the AV performs emergency braking. RoboTack achieves these results with only 14 camera frames for Disappear attacks, 32 frames for Move_Out attacks, and 24 frames for Move_In attacks. On the other hand, while the percentage of runs with safety hazards is still high, RoboTack is less successful in creating a safety hazard by attacking vehicles than attacking pedestrians. This is because LiDAR-based obstacle detectors cannot detect pedestrians at greater distances, and are only able to recognize larger obstacles such as vehicles. Although pedestrians are recognized by the cameras, the sensor fusion model does not increase spatial redundancy because only measurements from the camera-based pipeline are available for pedestrians.

4) Safety hijacker and impact on safety potential: This section compares the impact of using a safety hijacker (SH) on the resulting safety potentials and success rate of causing safety hazards. The results indicate that the timing of the attack chosen by the SH is critical for causing safety hazards with a high success rate. The comparison of the minimum safety potentials recorded for each scenario with attack vectors is shown in Figure 9 as box plots. In each subgraph, the green box on the left with the label "R w/o SH" means RoboTack with SH disabled (random attack timing). The orange box plot on the right with the label "R" means the full RoboTack malware is used to attack. We omit the Move_In attack vector because in those scenarios the attacks did not reduce the safety potential but only causing emergency braking. We can conclude from

Figure 9 that using SH significantly reduces the resulting safety potential and leads to more successful attacks. We summarize the improvement of the using SH over not using the SH as follows: a) DS-1-Disappear: 7.2x more accidents (31.7% vs 4.4%) and 4.6x (53.5% vs 11.6%) emergency brakings; b) DS-1-Move_Out: 6.2x more accidents (17.3% vs 2.8%) and 5.1x more emergency braking (37.3% vs 7.3%); c) DS-2-Disappear: 7.9x more accidents (82.6% vs 10.4%) and 2.4x more emergency braking (94.4% vs 39.4%); d) DS-2-Move_Out: 24x more accidents (84.4% vs 3.5%) and 14.8x more emergency braking (97.8% vs 6.6%); e) DS-3-Move_In: 1.9x more emergency braking (94.6% vs 50%); f) DS-4-Move_In: 1.6x more emergency braking (78.5% vs 48.1%). Again, we only compare the numbers of emergency brakings because Move_In attacks do not cause accidents. In summary, in the 1702 experiment runs, 851 with SH and 851 without SH, the experiment runs with SH result in 640 emergency brakings (75.2%) and 299 accidents (52.6%, excluding Move_In attacks), while without SH, these numbers are 230 emergency brakings (27.0%) and 29 accidents (5.1%).

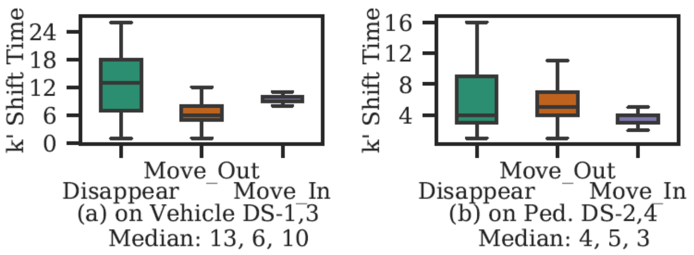


*Figure 10: Time Steps $k'$: the number of continuous attack time steps required to move the obstacle in or out*

5) Evading attack detection: Apart from high attack success rate, another crucial part of RoboTack is evading detections of the IDS. Figure 10 shows $K'$ for each attack

vector separately for pedestrians and vehicles. $K'$ is the shifting period in which the trajectory of the targeted obstacle is experiencing a "rapid change". After $K'$ the trajectory of the obstacle has been shifted to the desired location, and the trajectory will be maintained for an additional $K - K'$ time steps. $K'$ is generally much smaller than $K$. In those $K'$ time steps, we conclude that it is hard for the IDS to detect such abrupt changes because either $K'$ is small (a fraction of a second) for Disappear, or it is within one standard deviation of the characterized mean during the detector's normal operations for Move_Out and Move_In attacks, which require slightly longer attack durations than the Disappear attacks.

## 3.8 RoboTack: Conclusion

RoboTack is a smart malware that strategically attacks AV perception systems and results in safety hazards. Our work shows that when a malicious entity becomes familiar with the ADS, he or she can develop malware with scenario and timing awareness such as RoboTack to effectively target AV safety, by answering the questions of what, how, and when to attack. In particular, understanding when to attack allows RoboTack to attack at the most critical moment, and hence greatly improve attack success rate. We believe that RoboTack provides the following benefits: 1) Knowledge gathered from developing such malware can be used to explore flaws in an ADS system for future improvements. 2) The result from the attack can guide the development of countermeasures. The design of countermeasures is discussed in Chapter 8 of this thesis so we will only briefly discuss it here. One vulnerability in the perception system is the use of Kalman filter (KF) as tracker and sensor fusion model because KF cannot distinguish between malicious measurements and legitimate measurements—KF treats both types of measurements equally. A dynamic tracking and sensor fusion method that adaptively adjusts its parameters can help distinguish attacks from legitimate measurements and compensate for potential losses, and therefore mitigate most of these adversarial attacks.

# Chapter 4: DiverseAV: Low-cost Error Detection via Temporal Data Diversity

While autonomous driving systems (ADSs) are susceptible to malicious attacks, they are also susceptible to reliability issues, such as permanent and transient faults, because of heterogeneous compute architecture consisting of CPUs, GPUs, FPGAs, and ASICs. Faults such as a bit corruptions in the result register of CPU can lead to serious accidents if not detected in time. In this section, we introduce DiverseAV, [2] a low-cost, software-based, redundancy technique that improves the ADS' redundancy. We show in this work that DiverseAV can detect critical faults that lead to safety hazards, and achieves high accuracy in detecting permanent hardware and software faults without a full duplication of the hardware and software. In addition, DiverseAV is generally applicable to a variety of ADSs without the need to redesign the ADS extensively.

[2] DiverseAV is a paper in submission: *Exploiting Temporal Data Diversity for Low-cost Error Detection in Autonomous Vehicles*. The thesis author is a co-author (third author) of the paper due to his contribution in paper writing, method development, method evaluation, and experiment data analysis. Specifically, the thesis author designed and developed fault detection techniques for neural network weights-related faults and input data diversity analysis. The thesis author wrote the sections on the introduction and related works, methodology, experimental setup, sensor data diversity characterization, and evaluation and results. The rest of the work is mainly credited to Saurabh Jha and Timothy Tsai, the first author and the second author of the paper. The co-authors of the paper have granted permission to reprint its contents here.

## 4.1 DiverseAV: Overview

Autonomous vehicle (AV) uses an ADS for autonomous driving tasks. As introduced in Chapter 2, ADS consists of artificial intelligence (AI) and machine learning (ML)-based software to perform various tasks related to autonomous driving. These AI and ML-based softwares are computationally expensive. To ensure a real-time processing, ADS uses high-performance heterogeneous computing hardware, often consisting of CPUs, GPUs, FPGAs, and ASICs, for different tasks [3], [5]. However, the use of a large variety of hardware increases design complexity and hence makes the ADS more susceptible to permanent and transient faults. These faults can lead to serious accidents if not detected in time. For example, the faults might lead to an erroneous trajectory estimation of the pedestrian and incorrect decision of the ADS. Typically, commercial-grade ADSs use software and hardware duplication for fault detection [5]. One commonly used technique is duplication of the entire ADS (both software and hardware) and feeding the same sensor data to both duplicates. The outputs of the duplicates are compared bit-by-bit (assuming the ADS output is deterministic) for fault detection. While effective for fault detection, a full duplication of the ADS can be very costly and overly sensitive to faults. The latter could be problematic as a fault detection system that generates an overwhelming number of alarms can distract the human operator.

In this work, we propose DiverseAV a low-cost, lightweight, and effective software-based redundancy technique that exploits the temporal diversity of the sensor data for high-coverage software and hardware permanent fault detection, without significantly increasing the computational overhead. Going forward, the ADS software is referred to as the “agent”. DiverseAV has three key components listed below.

Independent agents: DiverseAV uses two independent and identical agents (ADS software) that are based on the same ADS software version and model. Here, independent means 1) the two agents maintain their internal states, 2) the two agents independently consume sensor input, and 3) the two agents make independent control decisions. At each time step, one agent instance is selected to run and drive the vehicle, and the other agent is selected to run at the next time step. In other words, the agent instances are run in a “round-robin” fashion. We refer to this as “time-multiplexed execution”.

Sensor data distribution and control fusion: DiverseAV runs the two agent instances in round-robin mode with a sensor data distributor, which is responsible for relaying the sensor data to the selected agent at each time step. For example, at $T = 2t$ agent 0 gets the sensor data $I_{2t}$, and at $T = 2t+1$, agent 1 gets the sensor data $I_{2t+1}$, where $t$ is a natural number. The sensor data distributor allows DiverseAV to exploit the temporal diversity of the sensor data. In the result section, we will show that though the sensor data of consecutive frames are semantically similar (i.e., camera images have slight differences visually between consecutive frames), they are representatively different at the bit level (i.e., very different when compared bit-by-bit at the same image location). Other than the default round-robin mode, DiverseAV also supports single-agent mode (running only one agent) or full duplication mode (in which two agents runs concurrently, getting the same sensor data at each $t$).

We denote the control output of two agents running in round-robin $O_{2t}$, and $O_{2t+1}$. The agents in the round-robin mode take turns to control the AV. We expect the control difference between $O_{2t}$ and $O_{2t+1}$ to be small since the semantic differences between the sensor data in consecutive frames should be small. Any abrupt change in control output between consecutive frames can be used for fault detection.

Fault detection: In the presence of a fault the outputs of the time-multiplexed agents can be very different as the fault will either impact only one agent (e.g. corrupted software states), or impact two agents differently (common hardware fault). The latter is because the two agents consume different sensor data that lead to different manifestations of the faults. Thus, data diversity between consecutive frames allows DiverseAV to detect faults by statistical comparison of the outputs. DiverseAV hence uses a sliding window anomaly detector to compare the statistical features (mean, median, etc.) of the outputs over time to avoid overly sensitive detection, as the outputs of the two agents under fault-free situation can be different from each other. The hyperparameters of the sliding window detector are learned from fault-free everyday driving. In addition, DiverseAV can detect faults that may lead to safety hazards in advance to provide a mitigation window for the ADS and human co-pilot. Such a detection technique is very different from duplicating the entire ADS and comparing the outputs bit-by-bit.

Low-cost and high-accuracy: DiverseAV is a black-box technique that duplicates the software portion of the ADS (agents) with the addition of sensor data distributor, fault detector, and control fusion, which makes DiverseAV usable on a variety of ADS. In addition, DiverseAV requires little compute overhead because the agent duplicates are running in round-robin mode. It is also highly viable and generalizable because it avoids software modifications to agents that are costly in terms of development and testing time, and it avoids hardware duplication as redesigning the platform can be costly. Finally, by exploiting the input diversity, DiverseAV provides high coverage of transient and permanent faults, with 0.87 precision and recall in detecting permanent GPU faults, and 0.99 precision and recall in detecting permanent neural network (NN) weight corruptions.

Overall, the key contributions of this work are the following: 1) We propose a novel redundancy technique, DiverseAV, for detecting permanent and transient hardware failures in ADS with high detection coverage and low compute overheads by exploiting sensor data diversity and time-multiplexed execution. 2) We implement DiverseAV with the state-of-the-art end-to-end ADS [20] using an advanced simulation platform [22] and evaluate the detection accuracy in a closed-loop environment. 3) We provide an empirical characterization of the temporal data diversity using real-world sensor data. 4) We perform an extensive experimental assessment on DiverseAV using GPU, CPU, and NN fault injections.

Our result in Section 4.6 shows that DiverseAV performs on a par with using the single agent, with the maximum deviation in various scenarios less than 50 cm. DiverseAV is also highly accurate and can detect permanent GPU faults with 0.87 precision and recall, and can detect permanent NN weight corruption with 0.99 precision and 1.0 recall, while only incurring 25% more overhead on CPU, 7% more overhead on GPU, 19% more system memory usage and 26.1% more GPU memory usage. The GPU permanent fault injection leads to more safety hazards than CPU permanent fault injection. Two percent (31 out of 1539) of the GPU permanent fault injections lead to a large deviation from the golden run, while none of the CPU permanent fault injections lead to trajectory deviation. However, 72.9% of CPU permanent fault injections lead to agent hangs (860 out of 1179 experiments). Finally, injection faults that corrupt a software state (NN weights) lead to a high number of safety hazards (42%, 1291 out of 3072 runs).

## 4.2 DiverseAV: Related Work

A large variety of techniques are used for fault detection and mitigation, ranging from low-level techniques such as parity and ECC protection to high-level techniques that involve duplication of the software and hardware.

Circuit hardening techniques are applied during hardware manufacturing to modify the operation parameters to improve reliability during usage. For example, one can tune the voltage and frequency of the chip to achieve maximum stability. But these techniques interrupt or reset the manufacturing process and induce significant cost.

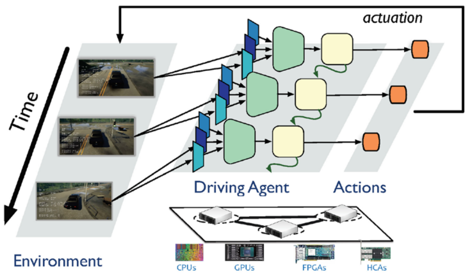
Hardware design techniques often include fault detection and mitigation at the circuit and architecture levels. These techniques include CPU instruction retry [39], ECC [40] included memory, hardware checkers [41], and parity codes [42]. Also, significant efforts have been spent on lock-step execution [43], single-core thread redundancy, or multi-core redundancy [44], [45]. Though effective, these implementations often require specific hardware designs of modification which induce additional cost due to chip area, power consumption, and synchronization overhead.

Software algorithm or enhancement techniques are for fault detection and mitigation at the software level with or without hardware support. Examples include algorithm-based fault detection [46], [47], software assertions [48], monitoring [49], and process duplication [50]. Software-based techniques have lower manufacturing costs than hardware-based techniques if no specific hardware is required, but because software-based methods often target specific pieces of software in the ADS stack, they are not universally applicable to the entire ADS. Moreover, these techniques require in-depth understanding of the software and redesign of the software to accommodate the redundancy, adding development overhead.

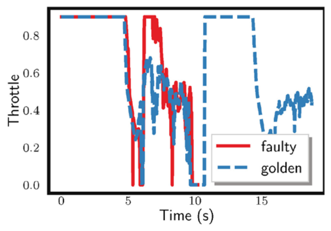
To put DiverseAV into perspective, DiverseAV is a lightweight, low-cost software-based redundancy technique that utilizes the diversity within the sensor data (input to the agents) to provide high-accuracy and high-coverage fault detection. The use of data diversity helps to tackle common cause failure (CCF), such as a failure in the hardware, because diverse input magnifies the faults in the case of CCF. Diversity can be introduced by using different versions of the program [51], by ordering the execution of instructions [50], by designing the program in a specific way [52], or by enforcing diverse data [53]. DiverseAV enforces diversity at the data level, and it ensures the agent's state diversity by time-multiplexing the sensor data (round-robin distribution) to the agents and running them on the same compute hardware, while maintaining resources usage by running only one agent at each time step. Furthermore, the agents are independent so that each agent maintains its states. DiverseAV can be further expanded to use more than two agents or to use different agents instead of pure duplication (N-version programming). Since DiverseAV runs each agent at half the original frequency (from the agent's perspective), it could impact the safety of the AV. However, a comparison with a single agent shows that DiverseAV has negligible effects on AV safety because vehicle motions are smooth and the sensor data between consecutive frames are semantically similar, so the two agents produce similar and smooth outputs for consecutive frames. DiverseAV is the first framework that leverages sensor data diversity for fault detection and mitigation for ADS.

## 4.3 DiverseAV: Background

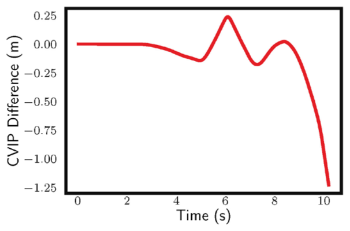
1) ADS: An ADS is a feedback-based control system that is responsible for driving the AV. At each time step, the ADS consumes sensor data and ultimately posts the control decision for the AV's controller to drive the car safely and comfortably. Two types of ADS are modular-based ADS and end-to-end ADS (used by our work). Figure 4 shows the architecture of an end-to-end ADS. The detailed architectures of both types of ADS are described in Chapter 2 so we omit them here. We use the state-of-the-art end-to-end ADS [20] for DiverseAV evaluation. At each time step, the end-to-end ADS consumes the sensor data and makes either the planning waypoints or control decisions using a single neural network (NN). The ADS we use outputs the planning waypoints and uses PID controllers [18], one for throttle, and one for steering, to convert waypoints into control decisions. An ADS is computationally expensive and often requires heterogeneous compute architecture consisting of CPUs, GPUs, FPGAs, and ASICs to achieve desirable performance for real-time driving [3], [5].



(a) ADS driving the car in the environment



(b) Impact of permanent fault on actuation



(c) Impact of permanent fault on safety

*Figure 11: ADS Hardware Fault Impacts on Actuation and Safety*

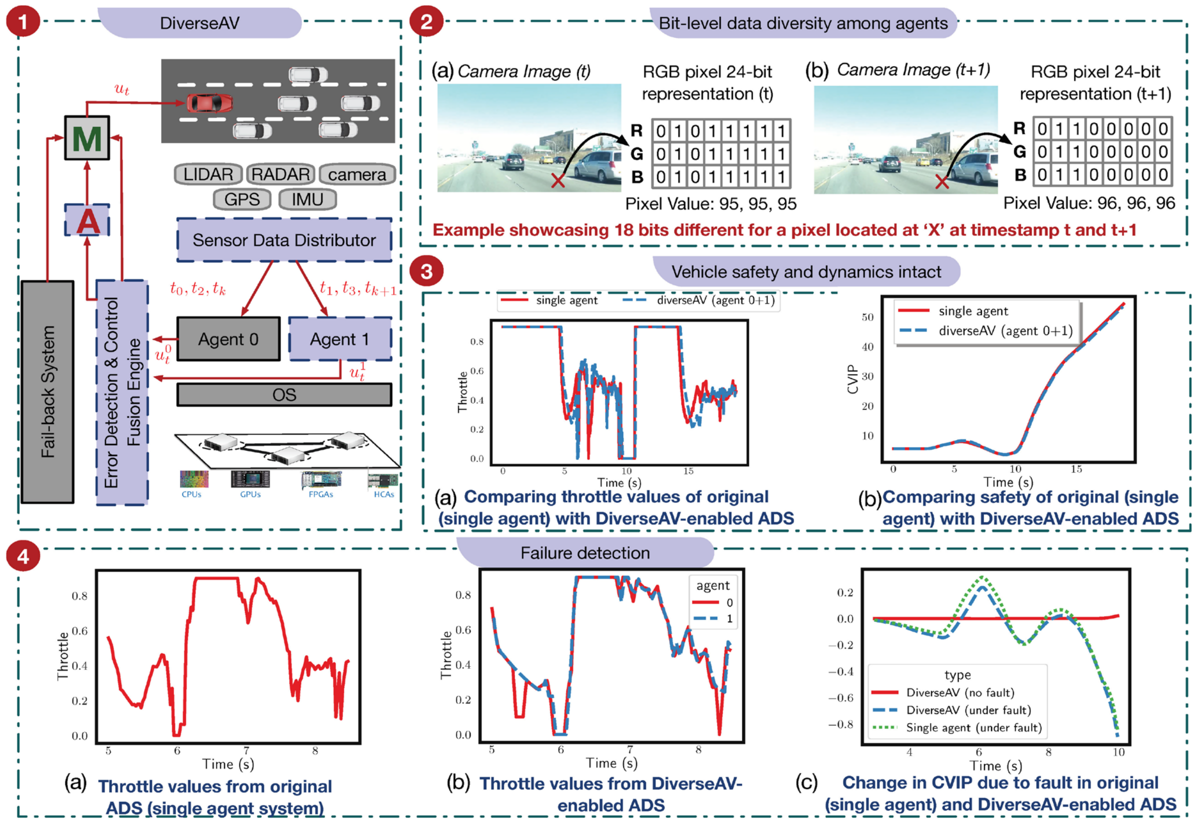
2) Impact of faults on safety: An ADS can experience various hardware and software faults, transient or permanent, and some of these faults can alter the control actuation from the ADS, impacting AV safety. Faults may lead to silent data corruption (SDC) and potentially propagate to the control decision that leads to safety violations. It is challenging to timely detect faults that lead to SDC as it might only manifest itself over time by accumulation. Timely detection of SDC requires costly solutions such as a full duplication of the ADS software and hardware for bit-by-bit runtime comparison. As an example, we show a GPU permanent fault that affects the throttle output of the ADS in Figure 11. Here the red line is the throttle output values in the presence of a GPU permanent fault, while the blue line is the throttle value of the fault-freed run. We can see the throttle output values are significantly different between the two runs, and the fault reduces the distance to the closest-vehicle-in-path (CVIP) in Figure 11. The faulty run ends with the AV colliding with the vehicle at the front at $T = 10$ due to the high throttle value early on. Hence, it is crucial to detect such a fault in time for timely mitigation (e.g. disengage the ADS and hand the control back to the human driver).

## 4.4 DiverseAV: Methodology

1) Design requirements: DiverseAV design allows us to address the following design requirements: (a) detection of transient and permanent faults, (b) workload independence of the ADS, (c) black-box plug and play solution, and (d) low cost. DiverseAV needs to detect transient and permanent hardware and software faults that are safety-critical with high accuracy and sufficient lead time. Since not all faults will lead to safety hazards, DiverseAV should not report all faults and thereby generate an overwhelming number of false alarms. DiverseAV design must be independent of the agent's design and should be usable on a large variety of ADS. DiverseAV should not require an extensive rewrite of the ADS software. Finally, DiverseAV should incur minimum overhead. Specifically, DiverseAV aims to be a completely software-based solution without the need for hardware duplication.

2) Design principles: DiverseAV uses two independent agents that are runtime instances based on the same ADS software and are executed time-multiplexed on the same ADS hardware. The outputs of the agent instances (or agents) are then inspected for fault detection. Time multiplexing execution of the agents has the following properties: 1) input data semantic consistency, 2) temporal sensor data diversity, and 3) high-coverage fault detection. Since each agent consumes data from consecutive time steps, and sensors are running at high frequency (e.g. 30 Hz to 60 Hz is typical for a camera), the data from consecutive time steps should be semantically similar, meaning

that the message conveyed by the data is similar. Time-multiplexed execution enables temporal data diversity at the bit-level. Though the semantics of the data from consecutive time steps should be similar (smooth transition), their bit-level representation can be very different. For example, for RGB image data, when a given pixel value changes from 95 to 96 for each color (a change in brightness), the data only changes by 1 on the face-value, but at bit-level, it differs by 18 bits out of the 24-bit RGB representation. Finally, time-multiplexing enables the detection of faults that propagate through the ADS stack and subsequently impact the AV's safety. The same hardware fault affects the agents differently due to sensor data diversity and independent agent states. Most of the time, the differences are significant enough and allow timely detection of a critical fault in the system. DiverseAV ensures that under normal execution, the safety of the AV is unaffected and each agent produces safe output to control the vehicle.



DiverseAV
$u_t$
M
A
LIDAR
RADAR
camera
GPS
IMU
Sensor Data Distributor
$t_0, t_2, t_k$
$t_1, t_3, t_{k+1}$
Agent 0
Agent 1
$u_t^0$
$u_t^1$
OS
Fail-back System
Error Detection & Control Fusion Engine
CPUs
GPUs
FPGAs
HCAs
Bit-level data diversity among agents
(a) Camera Image (t)
RGB pixel 24-bit representation (t)
R 0 1 0 1 1 1 1 1
G 0 1 0 1 1 1 1 1
B 0 1 0 1 1 1 1 1
Pixel Value: 95, 95, 95
(b) Camera Image (t+1)
RGB pixel 24-bit representation (t+1)
R 0 1 1 0 0 0 0 0
G 0 1 1 0 0 0 0 0
B 0 1 1 0 0 0 0 0
Pixel Value: 96, 96, 96
Example showcasing 18 bits different for a pixel located at 'X' at timestamp t and t+1
Vehicle safety and dynamics intact
single agent
diverseAV (agent 0+1)
Throttle
Time (s)
(a) Comparing throttle values of original (single agent) with DiverseAV-enabled ADS
CVIP
(b) Comparing safety of original (single agent) with DiverseAV-enabled ADS
Failure detection
(a) Throttle values from original ADS (single agent system)
agent
0
1
(b) Throttle values from DiverseAV-enabled ADS
type
DiverseAV (no fault)
DiverseAV (under fault)
Single agent (under fault)
(c) Change in CVIP due to fault in original (single agent) and DiverseAV-enabled ADS

*Figure 12: DiverseAV Overall Architecture Design*

3) Design overview: The design architecture of DiverseAV is shown in Figure 12-1. To enable sensor data diversity and time-multiplexed execution of the agents, we design and implement a sensor data distribution module and a control fusion module. Moreover, we design a fault detection engine to detect safety-critical faults.

The sensor distribution module takes sensor data $I_t$ at time $t$ and distributes the sensor data accordingly in a round-robin fashion, such that if agent 0 receives data for the current time step ($I_{2t}$), agent 1 receives data for the next time step ($I_{2t+1}$). Such design ensures that the agents get different input data and the data are semantically similar for safe autonomous driving while being significantly different at the bit-level representation. As shown in Figure 12-2, two images from consecutive time steps look very similar with minor transforms, but the pixel value's bit-representation is different for the same location at these two-time steps. We show later in the result section that on average, 8-bit out of the 24-bit representation of a pixel will change from time to time.

The control fusion module is responsible for aggregating the control decision of the agents and routing that to the AV's actuators. Depending on the types of ADS this control fusion of multiple agent instances can be challenging. For an end-to-end ADS like the Sensorimotor agent [20] used in this work, the sensor input and control decision are synchronized (lockstep execution). In this case, the design of the control fusion module is straightforward; since the input and output of the agent are synchronized, the module simply passes the output of the "lived" agent to control the vehicle. For a modular-based ADS, different modules can run at a different frequencies, so the two agents post control decisions asynchronously. The control fusion module can choose to either only pass one agent's decision to the actuators (a primary-secondary setup), or maintain a history of the control decisions and average the control decisions from the two agents and post that to the actuators. In this work, since the agent we used is synchronous, we naturally round-robin the agents' decisions to control the vehicle. Figure 12-3 shows that such round-robin control mode does not impact the AV safety during normal operations.

The fault detector is a module that compares the output of the two agents at two consecutive time steps and alerts the system if the outputs are faulty. In a deterministic system with full-duplication, the duplicated agents consume the same data and generate the same outputs in fault-freed conditions. The outputs from such a system can be compared directly bit-by-bit. This traditional approach is not suitable for ADS because the output of an AI-driven complex system is nondeterministic. Besides, full duplication of the system is costly. DiverseAV's fault detection engine can detect various faults using the input data diversity. The biggest challenge in designing the fault detection engine is robustness—accurate detection with high precision and recall, and timely—to detect the fault sufficiently before the safety hazard happens.

Even under normal execution, the outputs of the agent are different due to the differences in the input sensor data. We therefore use a sliding window-based fault detector and learn the statistic thresholds of output divergence between two agents using the golden runs. If the statistical divergence of the output between two agents is larger than the threshold, an alarm is raised to alert the human operator. This is shown in Figure 12-4, in which a fault causes throttle divergence between the two agents to increase. This increase in divergence affects the AV trajectory and the CVIP distance, resulting in a safety hazard. In the case of a single agent, the use of a PID controller smooths out the abnormal spike in the throttle from a single agent's perspective, so the fault is harder to detect in single-agent mode compared to DiverseAV.

The fault detector uses the following parameters: $\theta_{throttle}, \theta_{brake}, \theta_{steer}$, and $rw$. $\theta_{throttle}, \theta_{brake}, \theta_{steer}$ are thresholds for throttle, brake, and steering command differences between the two agents. An alarm is raised if the differences exceed the thresholds. The thresholds are conditioned on the vehicle's state as the tolerances are different under different states. Here, $\theta_{throttle}, \theta_{brake}$ depend on the velocity and acceleration of the vehicle, and $\theta_{steer}$ depends on the steering angle and the angular velocity of the vehicle. For example, a faster linear velocity may result in a higher threshold of the throttle and brake divergence. These thresholds are learned per statistical methods from training scenarios. We try different statistical methods on the sliding windows, including max, min, mean, median, and exponential mean, and find that max performs the best, so the thresholds are calculated by learning the maximum divergence between the two agents from the training data, conditioned on the vehicle states. The sliding window is used to smooth out the high transient divergence when the planning decision changes between two time steps because one agent will output a new control decision while the other agent has not yet made the same decision at those moments. The size of the sliding window $rw$ affects the sensitivity of the fault detector to such transient outliers. The size also affects the lead detection time. We vary $rw$ from 3 to 40, as our simulator, and hence the sensors run at 40 Hz (20 Hz for each agent).

## 4.5 DiverseAV: Experimental Setup

This section describes the experimental setup, including the end-to-end ADS, the simulation platform, the driving scenarios, the data collection procedure, the fault injectors, and the hardware platform used in this work.

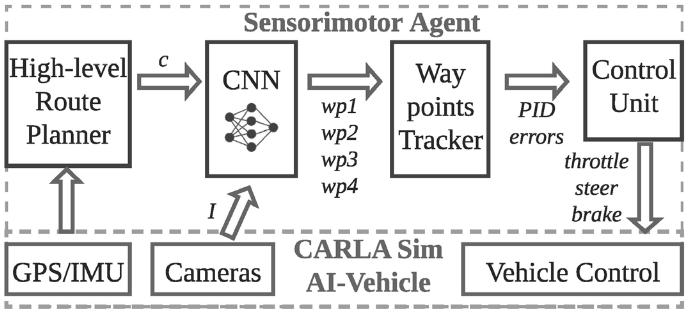


*Figure 13: Sensorimotor Agent*

1) End-to-end ADS: We use the state-of-the-art convolutional neural network (CNN)-based end-to-end autonomous agent proposed and pretrained by Chen et al. [20], referred to as the Sensorimotor agent. The agent is trained by imitation learning via expert demonstrations. The main component of the Sensorimotor agent is shown in Figure 13. The key components are high-level route planner, CNN, waypoint tracker, and control unit. First, the high-level route planner finds the next high-level destination to go based on the destination and the vehicle's current location from GPS measurements. Then, the CNN ingests the three front-facing cameras' data and performs vision-based planning by planning a local path around the obstacles, while following the lanes to the next high-level destination. The CNN outputs the local waypoints that the vehicle should follow in the immediate future. The waypoint tracker

tracks the waypoints from the CNN by calculating desired headings and the desired velocity. These desired values are then compared with the current vehicle state to generate heading and velocity errors for the control unit. The control unit uses two PID controllers to convert heading and velocity error into steering and throttle commands.

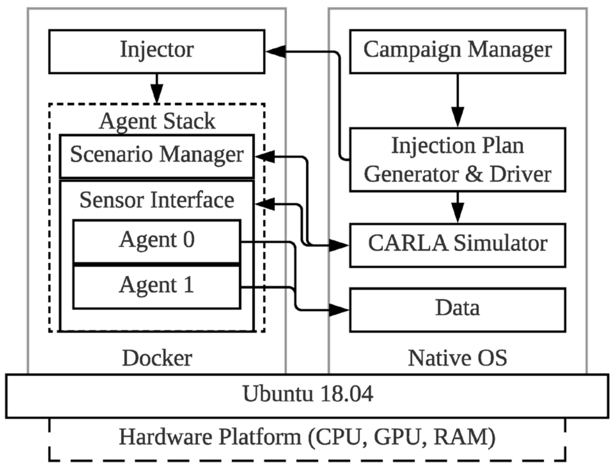


*Figure 14: Simulation Platform*

2) Simulation platform: The simulation platform architecture is shown in Figure 14. CARLA [22] simulates both urban and highway environments with various weather and road conditions. CARLA also simulates traffic, physics, and virtual sensor data for autonomous driving. This work uses three front-facing cameras, GPS, and IMU sensors as inputs to the Sensorimotor agent. The simulation runs at 40 Hz in synchronous mode, in which the simulator and the ADS are synchronized. The agent-simulator stack consists of two instances of Sensorimotor agents running in round-robin mode, a sensor interface, and a scenario manager. The two agents can be configured to run in round-robin mode (DiverseAV), duplicated mode (full software and process duplication), or

single-mode, in which only agent 0 will be active. The scenario manager is responsible for setting up the scenario events for the current run, such as the routes that the NPC vehicles should follow. Finally, the campaign manager reads the experimental configurations and launches the experiment accordingly.

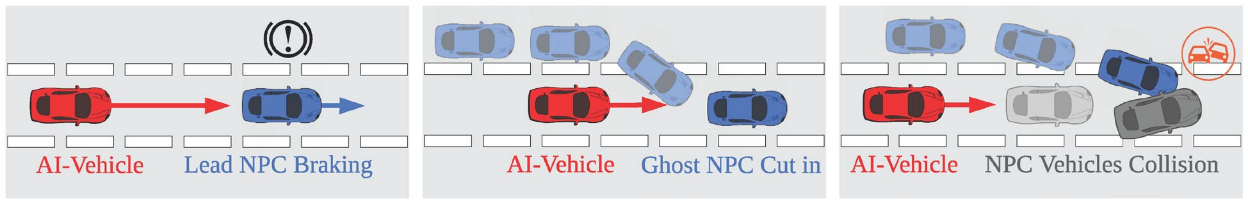


*Figure 15: Driving Scenarios: Left to right, Lead slowdown, Ghost cut-in, Front accident*

3) Driving scenarios: There are two types of scenarios: safety-critical scenarios and training scenarios. The safety-critical scenarios used in this work are shown in Figure 14. These are scenarios that are considered high-risk by the National Highway Traffic Safety Administration (NHTSA) [54]. The safety-critical scenarios are of a minute length and capture the most safety-critical moments during autonomous driving. We prepare the following safety-critical scenarios.

Lead slowdown: Shown in Figure 15-left, the AV (red) follows an NPC vehicle (blue) and maintains a distance of 25 m. The NPC then performs emergency braking and comes to a complete stop. The AV needs to infer this situation and apply the brake in time to avoid a rear-end collision with the NPC. The lead slowdown is a dangerous scenario because it leaves little time for the AV to react.

Ghost cut-in: Shown in Figure 15-middle, the AV is driving on the middle lane while maintaining its speed. An NPC vehicle approaches from the left adjacent lane and cuts in front of the AV with a small longitudinal margin. The AV needs to reduce speed immediately to avoid a collision. In this driving scenario, the NPC vehicle is not visible until it cuts in front of the AV, and hence it is also highly dangerous if the AV does not recognize this situation in time.

Front accident: Shown in Figure 15-right, the AV is following a leading NPC in the same lane, and another NPC vehicle tries to cut in but collides with the leading NPC. Both NPC vehicles suddenly change because of the impact and both come to a complete stop before moving on. The AV needs to recognize this situation and stop in time to avoid a collision. One challenge in this scenario is that the motion of NPC vehicles suddenly changed after the collision. Although this incident is rare, the AV might not recognize the abrupt change in trajectories of the leading vehicles and thus might output the incorrect decision.

The training scenarios are scenarios used for training. These are long scenarios with 10 min of autonomous driving. These scenarios are selected from the 2020 CARLA Challenge, with fault injections disabled. The long scenarios consist of important driving tasks such as lane keeping, turning, lane-changing, and handling of intersections. We later show in Section 4.7 that the fault detector parameters can be learned from these long scenarios, which are used as references for normal everyday driving without faults.

The learned parameters achieve high precision and recall in detecting safety-critical faults. The long scenarios include different driving environments including urban, highway, and a hybrid of both.

4) Data collection: We collect the control decisions, including the throttle, steering, and brake value from the two agents, and the AV trajectories and states at each time step. We also collect the distance to the CVIP, steering, and velocity error from the waypoint tracker and the predicted waypoints from the CNN. In addition, the scenario manager summarizes driving statistics including route completion percentages and the number of collisions. We also collect logs from the system, fault injectors, and the agent-simulator stack for experiment and fault injection status analysis.

5) Fault injection: We consider the following fault models: 1) instruction-level bit-flip models for CPUs (via PinFI [55], [56]), and GPUs (via NVBitFI [57], [58]) and 2) weight perturbation of the neural network (via PyTorchFI [59]). We focus less on transient hardware faults because those faults are short-lived and affect only a single agent, so they can be easily detected. Permanent faults are long-lasting and hence they affect both the agents, which are more dangerous to autonomous driving.

GPU fault injection: We use NVBitFI for GPU fault injection. NVBitFI simulates the consequences of underlying permanent fault at the instruction level of the GPU, by corrupting the instruction result register of a selected opcode for all dynamic instances of that opcode. A dynamic instance of an opcode is that specific opcode with the operand, and instructions with the same opcode but different operands are counted as different dynamic instances. This fault model affects all dynamic instances with the same opcode during execution. NVBitFI corrupts the result register by XOR-ing the original value with a mask. In our case, the mask is fixed and user-defined for all injections. The mask we choose simulates random multiple bit-flips. The ISA of an NVIDIA Titan Xp GPU has 171 opcodes, and for each opcode, we perform injections on all three safety-critical scenarios and repeat each opcode injection three times to capture any non-deterministic behaviors. In addition, we run 50 fault-free experiments for each scenario as fault-freed golden runs. We make sure that the injections only affect the agents but not the simulator.

CPU fault injection: CPU injections are done using PinFI. The fault model and injection procedure are similar to NVBitFI and we inject all of the 131 Intel opcodes used by the ADS software. Again, we make sure that the injections only affect the agents but not the simulator.

Software fault injection: In an end-to-end ADS the NN is the most critical, complex, and computationally expensive component of the ADS. Since AV is an IoT device and NN weight update is done by over-the-air (OTA) update, it is possible that the NN weight is corrupted during weight transfer. Besides, the weight can also be corrupted when it is transferring from the disk to system memory, or from system memory to device memory. Therefore, we inject faults directly into one of the agent's

NN by perturbing the NN weights using PyTorchFI. This simulates the fault model of a corruption of the software state due to a transient fault, and the corruption disappears when restarting the software or the system. A transient fault in the software will likely impact only one of the agents because the agents are independent. We only inject faults into one agent and one weight of the 1024 weights in the last layer for all three safety-critical scenarios with PyTorchFI. PyTorchFI corrupts the weights by XOR-ing a weight with a random mask to simulate multiple bit flips of the weight. For each driving scenario, we ran 50 fault-free experiments as the golden run for each scenario. NN injections do not need repetition for their deterministic behaviors.

6) Hardware platform: the GPU and CPU injection experiments are run on platforms with Intel Xeon E5-2699v4 CPU, 64 GB of RAM, and an NVIDIA Titan Xp GPU. The NN weight injection experiments are run on platforms with a Ryzen 5 2600 CPU, 16 GB of RAM, and an NVIDIA Titan Xp GPU. All fault injections use the same version of ADS and CARLA Simulator.

## 4.6 DiverseAV: Evaluations and Discussion

1) Characterizing sensor data diversity: DiverseAV leverages sensor data diversity to achieve a high-coverage of fault detection because the fault manifestation differs with different input data. We characterize the sensor data diversity of both the simulated sensor and the real-world sensor data.

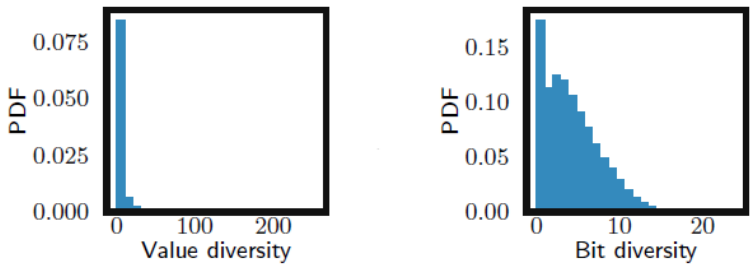


*Figure 16: Value and Bit Diversity of Simulated Sensor Data*

The simulated sensor data are collected from the CARLA simulation on the long driving scenarios. We evaluate the value diversity (shown in Figure 16-left)—the difference in color intensity (0—255 for each of the R, G, and B color of a pixel is represented as an 8-bit unsigned integer), calculated per color channel and per pixel location between two consecutive camera frames from the same camera. There are three cameras, facing front, front-left, and front-right, in our simulation. We also evaluate the bit diversity (shown in Figure 16-right)—the number of bit differences in the 24-bit color-representation, calculated per corresponding pixel location between consecutive camera frames from the same camera. The distribution of value diversity has a mean of 3 and the 90th percentile of 14, with 255 being the maximum possible diversity between two pixels. In contrast, in the distribution of bit diversity, the mean is 5 bits changed in a 24-bit pixel representation, with the 90th percentile of 9 bits. We can see that even though the value diversity is small, the bit-level diversity is large. The latter is important because the computer hardware operates on bits (representation of the values) instead of the values themselves. As a result, a larger bit diversity results in a more diverse manifestation of the same permanent faults on two different agents, which allows the fault detector to detect common-caused-failures (CCFs).

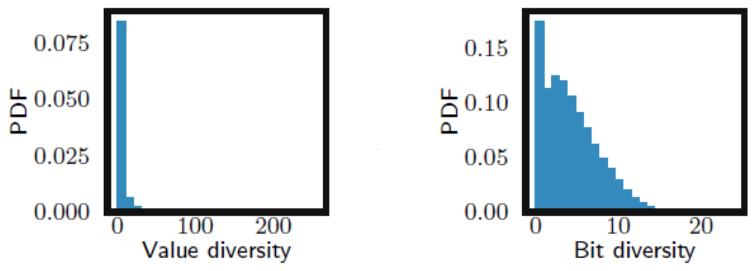


*Figure 17: Value and Bit Diversity of Real World Sensor Data*

The real-world sensor data have larger diversities in both value diversity and bit diversity time step by time step because real-world sensor data are noisy which further increases the diversity. We use KITTI Raw Data [60] dataset for diversity characterizations on real-world sensor data. The dataset consists of LiDAR, camera, IMU, and GPS sensor data collected at 10 Hz from real-world driving. We characterize the camera sensor data diversity using the same method we use for simulation data with a total of 9391 camera frames. As shown in Figure 17, the value diversity of the real-world camera data is 5 at the 50th percentile and 58 at the 90th percentile, while the bit diversity is 8 bits at the 50th percentile and 13 bits at the 90th percentile (24-bit RGB pixel representation). In other words, while the value diversity is seemly small, the bit diversity is large. We also characterize floating-point data such as IMU and GPS data and find that the bit diversities at the 50th and 90th percentile are 11 bits and 15 bits out of the 32 bit floating-point number.

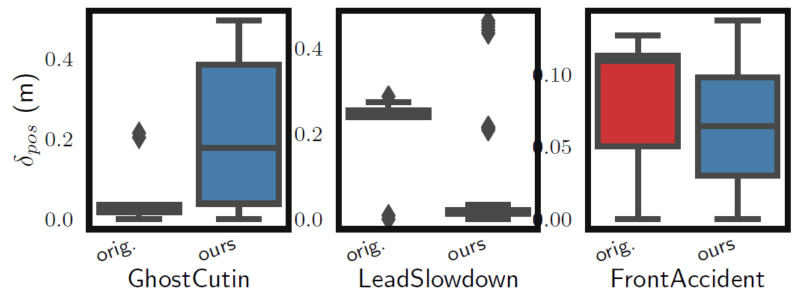


*Figure 18: Impact of DiverseAV on Safety Potential*

2) Characterizing the impact of DiversAV on AV safety: One key design requirement of DiverseAV is that it should not affect the AV's safety due to the time-multiplexing execution of the agents situations. We characterize the impact of DiverseAV on AV safety by comparing its trajectory with the original single-agent ADS (baseline) setup. The trajectory of the AV is a list containing the global position of the AV in the simulation environment over time. We use $\delta_{pos}^{E,B} = max(traj^{E} - traj^{B})$ to denote the maximum divergence between the DiverseAV and the baseline trajectories. Figure 18 shows the boxplot of the $\delta_{pos}^{E,B}$ of the three safety-critical scenarios, calculated using 50 runs for each setup. We can see the deviation is less than 50 cm for all safety-critical scenarios. Therefore, we conclude that while DiverseAV affects AV's trajectory, it mimics closely the original single-agent setup and will not cause a safety hazard. In fact, in none of the runs has DiverseAV caused any safety violation.

*Table 3: Fault Injection Summary. All experiments use in round-robin agent mode. DS: driving scenario, LSD: lead slow-down, GC: ghost cut-in, FA: front accident, # Trajectory Violations: number of experiment with trajectory violation that are accident-free, # Accidents: number of experiments with crashes*

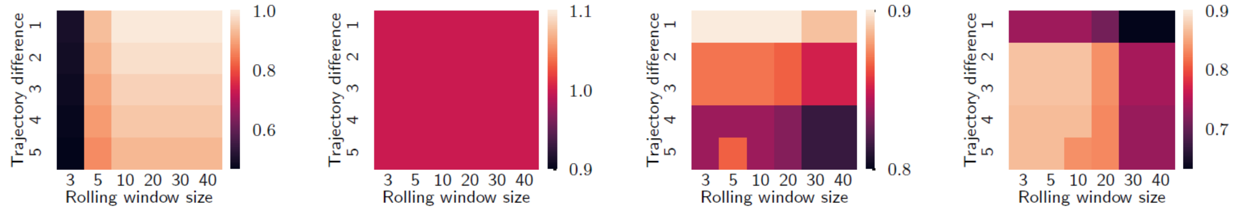
| FI Target | DS | # Golden | # FI Exp (Hang/Crash) | # Accidents | # Trajectory Violations |
|---|---|---|---|---|---|
| GPU | LSD | 50 | 513 (83) | 3 | 9 |
| GPU | GC | 50 | 513 (83) | 14 | 2 |
| GPU | FA | 50 | 513 (81) | 0 | 3 |
| CPU | LSD | 50 | 393 (287) | 0 | 0 |
| CPU | GC | 50 | 393 (286) | 0 | 0 |
| CPU | FA | 50 | 393 (287) | 0 | 0 |
| Neural Network | LSD | 50 | 1024 (84) | 378 | 55 |
| Neural Network | GC | 50 | 1024 (100) | 226 | 187 |
| Neural Network | FA | 50 | 1024 (66) | 393 | 52 |

3) Characterizing fault in ADS: Table 3 summarizes the experiments for each fault injection method per critical scenarios and their outcomes. In total there are nine combinations from three fault injection methods and three critical scenarios. We also use three additional driving scenarios (Route02, Route15, and Route46) for training the fault detector (finding thresholds). These training routes have zero accidents or trajectory violations so they are not mentioned in the table. For each of the campaigns, we ran 50 fault-freed (golden) runs to characterize the simulation's nondeterminism, understand the impact of the fault on the AV's safety, and test the threshold learned from the training scenarios under fault-freed conditions. The impact of the faults on the AV's safety is quantified by two metrics: number of trajectory violations, and number of accidents. During a run, the trajectory is violated if $\delta_{pos}^{E,B} > 2$ (i.e., the maximum divergence between the experiment run (E) and the baseline (B) exceeds 2 meters). We vary $\delta_{pos}^{E,B}$ to reveal the detection capability of our proposed design. The trajectory of B

is the average of the trajectories from all golden runs for the same scenario. The next metric is the number of experiments that have accidents; the experiment terminates if an accident happened, and the accident incident is reported by the scenario manager to the logs.

For hardware fault injections (FIs), CPU FI results in the highest percentage of hangs and crashes with 72.9% of the experiments (860 out of 1179 runs), but zero trajectory violations or accidents across all runs. Hangs and crashes can be detected promptly by the system, and the system can quickly alert the human operator to take over the vehicle. Those hangs and crashes are expected because FI into CPU instructions will corrupt the control flow and the memory accessing address of the program, resulting in hangs or crashes. On the other hand, we see fewer crashes with GPU FIs but a higher percentage of safety violations. Only 16% of GPU FIs result in hangs or crashes (247 out of 1539 runs), 1.1% of GPU FIs end in accidents, and 0.9% of GPU FIs end with trajectory violations. In some incidents, GPU FI causes silent data corruption (SDC) that is undetectable by DiverseAV and significantly impacts the AV's trajectory, leading to safety violations.

Software fault injections (FIs) resulted in the highest percentages of accidents (32.4% which is 997 out of 3072 runs) and trajectory violations (9.6% which is 294 out of 3072 runs), and the lowest percentage of hangs and crashes at 8.1%. This is expected because the FI directly corrupts the weights that affect the NN's output, which is the local waypoints for the AV to follow.

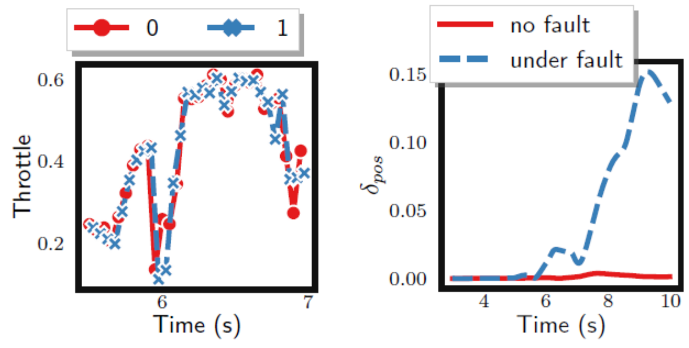


**(a)** NN Weight FI Precision. **(b)** NN Weight FI Recall. **(c)** GPU FI Precision. **(d)** GPU FI Recall.

*Figure 19: Heatmap, precision and recall under different hyperparameters*

4) Characterizing error detection capabilities: Two other key requirements of DiverseAV are high accuracy and high coverage fault detection. At the very least, DiverseAV should aim to detect all safety-critical faults—faults that lead to accidents or trajectory violations. In addition, DiverseAV should not raise an alarm for golden runs because those runs are fault-freed. The fault detection capabilities are evaluated in terms of precision, recall, and lead detection time. Moreover, we vary the two hyperparameters, $rw$ and $td$, to find the best hyperparameter settings. The term $rw$ is the size of the sliding window, and $td$ is the threshold for trajectory violations (instead of just 2 m, $td$ can range from 1 to 5 m). We evaluate DiverseAV using only experiments with GPU FI and NN FI because a majority of NN FI cause hangs and can be easily detected by the system. We evaluate DiverseAV on both the safety-critical scenarios (including golden runs) and long scenarios and observe if DiverseAV raises false alarms on fault-freed runs.

GPU FI: Figure 19-1 and 2 show the heat maps of precision and recall of the fault detector on detecting GPU FIs using different hyperparameter configurations. The best performance is obtained by using $rw = 3$ and $td = 2$, with a precision and recall of 0.87. DiverseAV does not generate alarms on any of the golden runs and flag runs



**(a)** Throttle values for time-multiplexed agents 0 and 1. **(b)** Trajectory difference.

*Figure 20: DiverseAV Detection Missed Case*

for which the two agents' outputs are noticeably different. The precision and recall are not 1 because there are cases of false negative and false positive. False-positive cases are cases in which the two agents' outputs differ but do not result in safety violations. False-negative cases are more interesting, as these are cases with safety violations but the output of the two agents are similar throughout the experiments. One such case is shown in Figure 20 and the targeted GPU opcode is 44 (IMADSP), which performs integer exact multiply-add operations. We can see from the figure that though there is a safety violation, the output of the two agents is closed. Understanding the cause is not the focus of this work, so we omit it here. Figure 21-right shows the lead detection time for the fault detector on GPU FIs. We can see that DiverseAV can detect the faults with a lead time significantly greater than 1 sec, which leaves sufficient time for the human driver to take control of the vehicle.

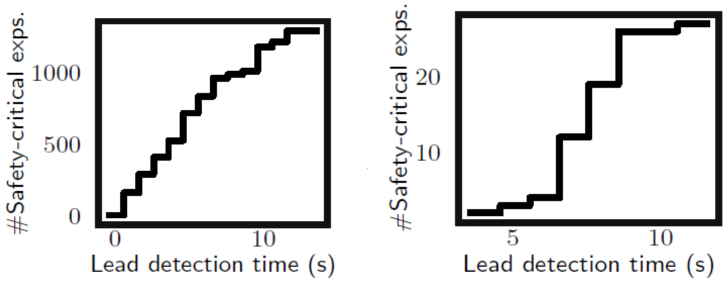


*Figure 21: Lead Detection Time, left NN Weight FI, right GPU FI*

NN weights FI: The NN FI by its nature only affects a single agent so the output of that agent, if affected, is significantly different from the normal agent. Interestingly, the AV is found to be safe in many of the runs, despite the fact that the fault should impact the output of the agent directly. This could be attributed to the compensation from the good agent, or insignificant weight values. For NN FI fault detection, we relax the fault detection model so that the threshold parameters for throttle, brake, and steering, $\theta_{throttle}, \theta_{brake}, \theta_{steer}$, are constantly used for detecting faults for NN FI. Similar to parameters conditioned on the AV's state, we learn these parameters from the training scenarios and verify them using the golden runs. The precision and recall heat maps of choosing different $rw$ and $td$ are shown in Figure 19-3 and 4. The best performance of precision and recall of 1.0 is achieved by setting $rw = 10, td = 1$. The fault detector robustly detects the faults for other settings of $rw$ and $td$ as well. Figure

21-left shows the lead detection time for $rw = 10, td = 1$. We observe that the lead detection time is zero for 2 cases, but for over 93.8% of cases, the lead detection time is greater than 1 sec, given the ADS sufficient for mitigation. Again, DiverseAV generates an alarm for none of the golden runs.

*Table 4: DiverseAV Performance Overheads*

| | **CPU %** | **GPU %** | **RAM** | **VRAM** |
|---|---|---|---|---|
| Single Agent | 4% | 14% | 2258 MB | 757 MB |
| Dual Agent | 5% | 15% | 2689 MB | 955 MB |
| Duplicated ADS (Estimation from Single Agent) | 8% | 28% | 4516 MB | 1514 MB |

5) Resource overhead: The final key requirement of DiverseAV is low-cost. We measure the resource overhead of the DiverseAV during fault-freed execution on training scenarios and compare it with the single-agent baseline. As shown in Table 4, DiverseAV increases the CPU and GPU usage marginally and consumes significantly less compute power than the full duplication execution. DiverseAV does increase both system memory and GPU memory usage, by 19% and 26% respectively, due to additional states when running two agents. The memory did not double because DiverseAV only duplicates the ADS agent and part of the memory is used by shared libraries and infrastructures. Compared to a full duplication of the entire autonomous driving software, DiverseAV reduced computational overheads by 46.4% for the GPU and 37.5% for the CPU, without the need for duplicating the hardware.

## 4.7 DiverseAV: Conclusion

In this work, we proposed DiverseAV, a low-cost, high-accuracy, and high-coverage redundancy technique for autonomous driving agents that leverages the temporal data diversity of the sensor data. DiverseAV runs the agent duplicates in time-multiplexed mode which significantly reduces compute overheads compared to a full duplication of the ADS software while providing high fault detection coverage. Our result shows that DiverseAV can detect various hardware and software faults that affect one or both of the agents without the need to fully duplicate the ADS. DiverseAV treats the ADS as block-box and only requires an additional sensor distribution module and fault detection module, without requiring extensive modification or redesign of the ADS software. This property makes DiverseAV generalizable to a large variety of ADS.

# Chapter 5: HPC Storage System: Overview

Chapters 5 and 6 and describe the works related to high-performance computing (HPC) storage systems. A storage system of the HPC analogs to the storage device of a personal computer that stores system and user data but at a much larger scale. An HPC storage system must meet the desired quality-of-service (QoS) requirements in terms of latency and throughput because the compute nodes in an HPC are storage-less with their performance bounded by the storage system. A fault in the HPC storage system may propagate through the system and affects a wide range of applications. The most critical faults cause system-wide outages (SWOs) in which the entire HPC storage system becomes unavailable. When that happens, the applications running on the HPC crash, and hundreds and thousands of compute hours are wasted, incurring heavy financial losses. However, real-time system failure localization and diagnosis in the HPC storage system are challenging because 1) a large number of interconnected components result in many failure modes, making it hard to pinpoint the exact location of the failure, 2) failures can be transient and masked by the system redundancy so there is no effect on the system's operation, or they can propagate through the system and have different manifestation at different layers of the system, 3) a large variety of user applications stress the system components differently and can cause different failure modes, and 4) it is hard to distinguish component failures from resource contention because of their similar impact on performances.

Our work Kaleidoscope focuses on real-time failure localization and diagnosis of the HPC storage system while tackling the aforementioned challenges via smart monitoring, probabilistic graphical model (PGM), and natural language processing. We will start by describing the key components of a typical HPC storage system using the Cray Sonexion-1600 storage system in NCSA Blue Waters Supercomputer as an example. Cray Sonexion-1600 storage system uses Lustre as its filesystem and has a peak throughput of 13.3 petabytes per second. Lustre is a high-performance peta-scalable distributed filesystem tailored for HPC applications and supports POSIX compliant interface. A typical HPC storage system forms a server-client interaction model with the compute nodes via networks and process file queries from the compute nodes.

## 5.1 Client

The HPC storage system serves the compute nodes that run the user applications. The compute nodes are referred to as clients. Clients connect to the storage servers via network and send file queries to the storage servers. Upon accessing a file, the client first queries the metadata servers for file metadata, and the client has the permission to access the file; the client then queries the object storage servers for the actual file content. Depending on the dynamic routing algorithms of the network, a client has multiple pathways to connect with the storage servers and accesses the files.

## 5.2 Metadata Server (MDS)

The metadata server hosts the metadata of the files stored on the file system. The files are stored on the filesystem as objects, and the metadata of the file includes filename, inode, and list of file object's locations on the object storage device. The metadata server stores the file metadata on metadata targets (MDTs), which are arrays of drives configured in RAID for speed and redundancy. The MDS is the starting point of accessing a file as the file can only be accessed with the metadata. Due to constantly searching for file metadata, the MDS accesses comprise many small and random transactions and hence the target devices are designed accordingly. For example, in Cray Sonexion-1600, the MDS consists of 14 450 GB SAS drives for enterprise-grade reliability and performance in RAID 1+0 mode, and one 100 GB SSD for filesystem journaling and journal-replay (imperative recovery). The MDS is configured in active-active or active-passive (in the case of Blue Waters) pairs for improved reliability.

## 5.3 Object Storage Server (OSS)

The object storage server (OSS) handles file inquiries from the clients and stores the actual file objects. File objects are blocks of fixed size stored on the storage devices on OSS known as object storage targets (OSTs) and a file is stored as stripes across multiple OSTs to maximize the performance and redundancy. OSSes are connected in active-active or active-passive configuration pairs and in the former configuration can serve the file inquiries and replace the faulty OSS if one of them fails. The OSS has much more storage space than MDS and on Cray Sonexion-1600; each OSS-pair has 80 disks of 2 TBs each organized in units of 8+2 configured with RAID 6. The OSS also includes "hot swap" disks to swap out faulty disks on the fly to improve redundancy.

## 5.4 Management Server (MGS)

The management server or MGS handles platform-specific configurations and stores book-keeping information about the components. The data are stored on local storage devices named management data targets (MGTs). MGS is responsible for storage system monitoring and imperative recovery in case of a system failure. In Cray Sonexion-1600, the MGSs are configured in active-active pairs for improved redundancy.

## 5.5 Interconnections

The individual server of the storage system is connected by an internal storage network. For example, the Cray Sonexion system's interconnection is the Infiniband network. This network is different and independent from the network used by the compute servers or the interconnections between the compute servers and the storage servers. The health states of the storage network directly impact the efficiency of the storage servers, and a network outage due to network component failures or congestions can cause a serious system-wide outage of the storage system. The internal storage network likely has load balancing and dynamic routing, so the route taken by a file access request can be dynamic.

## 5.6 Reliability Features

The MDS, OSS, and MGS are all configured in high-availability pairs or HA-pairs. By connecting two servers in pairs, one server in the pair can take over the other server's job and continuously serve the clients in the case of a server failure within the HA-pair. The HA-pair can be configured as either active-active, in which both servers serve the client, or active-passive, in which only one serves the client while the other serves as a backup. When a failure happens, the monitoring system running on the MGS detects such failure, kills the failed server, and configures the system so that it uses only the other/backup server to serve the client. The HA-pair also enables imperative recovery to recover a failed-recovered server using the healthy server and updates the failed-recovered server with the missing file transactions.

Imperative recovery is triggered on the failed server by the MGS. The imperative kills and reboots the failed server, and updates the server availability table for the client to avoid the client connecting to the faulty server. The failed server, after reboot, also replays the transactions on the healthy server to recover the lost transactions. These procedures are transparent to the client and perform solely on the server side of the storage system. However, depending on the duration of the outage or the workload of the storage system, imperative recovery can affect system performance and take a prolonged time to complete.

Another redundancy feature is the RAID array, which can be used for disk-level redundancy by pairing up individual disks making some disks in the pair as backups. In a RAID 1+0 configuration, $K$ disks are configured as a RAID 1 stack and multiple of these RAID 1 stacks are configured as RAID 0 striping for speed. In a RAID 6 configuration, the disks are configured as striping with dual parity, with two parities per drive. Such a configuration has high fault tolerance and can withstand simultaneous two-disk failure in a single array.

# Chapter 6: Kaleidoscope: Live Forensics for HPC Systems: A Case Study on Distributed Storage Systems

This work presents Kaleidoscope [3] [7], a real-time HPC storage system failure recognition framework that uses domain knowledge and machine learning techniques for HPC failure detection, localization, and root cause diagnosis. Kaleidoscope not only detects the failures, but also infers the failure modes, and finally identifies the possible failure components. Kaleidoscope is deployed on the Blue Waters supercomputer and evaluated with more than two years of monitoring data from production. The evaluation shows that Kaleidoscope can localize 99.3% of the failures and pinpoint 95.8% of the 843 known, real-world production issues, with negligible compute overheads.

## 6.1 Kaleidoscope: Overview

The HPC storage system is a high-performance, large-scale system that could experience a wide range of failures due to its complexity. These failures include

[3] Kaleidoscope is accepted and published as the paper: *Live Forensics for HPC Systems: A Case Study on Distributed Storage Systems* [7] in the *2020 International Conference for High-Performance Computing, Networking, Storage, and Analysis*. The paper was selected as a finalist for both the Best Paper and the Best Student Paper awards. The thesis author is a co-author (second author) of the paper due to his contribution in paper writing, method development, method evaluation, and experiment data analysis. Specifically, the thesis author helped the development of the fault diagnosis model by telemetry data analysis, a core component in Kaleidoscope. The thesis author wrote the sections on methodology, evaluation and results, and operational experiences. The rest of the work is mainly credited to Saurabh Jha, the first author of the paper. © 2020 IEEE. Reprinted with permission.

reliability failures that result in hangs and crashes and resource overload failures that result in resource scarcity. These failures often have similar impacts on the client side—a slowdown or inability to access the files. However, the causes of these failures can be very different, ranging from component failures such as a failed disk or network switch, or application abuse such as a rogue program frequently accessing the files. Different causes require different mitigation plans (restarting or isolations for component failures and throttling for resource overload failures) to be carried out in time to mitigate the failures. The inability to mitigate these failures can lead to system-wide outages (SWOs) that seriously impact the applications running on the system. Thus, to carry out the correct mitigation plan in the presence of a failure, the monitoring system needs to detect and localize the failure and then diagnose the failure in real time.

Kaleidoscope is designed to address the aforementioned issues. Kaleidoscope uses machine learning (ML) techniques for failure localization and failure diagnosis by using existing monitoring infrastructure. We demonstrate Kaleidoscope's effectiveness on the petascale distributed storage system of the NCSA Blue Waters Supercomputer and show that we can achieve failure localization and diagnosis in real time. We focus on the storage system because NCSA system administrators reported that in 2018 64.4% of the loss in compute hours (32 million core hours) were due to storage-related problems. Further, the problem is expected to get worse in the future exascale systems because of increases in both components and system scales.

ML-driven methods: Kaleidoscope is an ML-driven failure recognition framework that uses multi-modal telemetry data that provides comprehensive temporal and spatial information system-wide. These data are collected by actively polling the system components using pings or heartbeats, or passively aggregating performance measurements based on server loads. The telemetry data are often noisy and only contain partial information because of asynchronous collection, failure propagation, and non-determinism in the system due to routing and load balancing. These telemetry data need to be aggregated and analyzed as a whole for accurate diagnosis of the failures, and singular analysis can lead to false positive or false negative detections. However, analyzing these data manually is not an easy task because the data are large scale (a few TBs per day) and recorded in system logging format. Manually analyzing the data often causes significant delays in failure identifications and misses the timeframe for mitigation, leading to system outages. Kaleidoscope uses ML methods to automatically infer component health states and the root causes of the failures using these noisy data in real time, providing real-time failure localization and diagnosis. Kaleidoscope differs from existing works as it 1) jointly diagnoses and distinguishes between resource overload failures and reliability failures, 2) focuses on a wide range of system components instead of a single type of component such as storage devices, and 3) recognizes failures without the need for explicitly labeling the data for training.

Approach overview: Kaleidoscope consists of hierarchical domain-guided interpretable ML models: a failure localization model for component failure localization a failure diagnosis model for identifying the failure mode and root cause of the failure.

The failure localization model uses a probabilistic graphical model (PGM) and I/O path tracing data (Store Pings) for state estimation of system components. System components include metadata servers, object storage servers, network components, and clients (compute nodes). The I/O path tracing data provide information on component responsiveness and availability on the route taken by the storage accessing request, issued from the clients (compute nodes) to the actual disks on the storage servers. Successfully serving the request requires every component on that route to be available and healthy. Since each of these I/O path tracing data provides only a partial view (one path/route) of the system state, the failure localization module jointly evaluates the path tracing data and infers the system's health state as a whole.

PGM is used to analyze the noisy path tracing data by expressing the interrelationships between the components and the paths as statistical relations. The PGM is modeled to capture the path uncertainties due to load balancer and dynamic routing protocol via statistical relations. The failure states of the components are modeled as hidden random variables of the PGM in which their values need to be inferred from the observed variables—the path availability, indicated by the telemetry data. The connectivity of the PGM captures the interrelationships between the random variables and it is derived from the path tracing monitors, the component failure distributions, and the system topology. Using this setup, the PGM can jointly evaluate different sets of path tracing data, which is a form of data aggregation to reduce the noise within the data.

To collect the path tracing data, Kaleidoscope uses a set of metrics called Store Pings to evaluate the latency of a path from a client to the disks. Store Pings are low-cost, low-overhead monitoring probs that are issued periodically from the client to the storage server to capture path latency. At each time step, the path of the Store Pings can be configured beforehand to cover a specific path in the system. Store Pings are placed strategically in the system to achieve maximum coverage by minimum overheads.

The failure diagnosis model is a domain knowledge-embedded statistical model that identifies the failure mode and the root cause of the failure. The failure diagnosis model achieves these by using the component's telemetry data which includes performance metrics and logs as well as the health state of the components estimated by the failure localization model. The failure diagnosis model utilizes local outlier factor analysis [61], which is an unsupervised method model for anomaly detection based on local density.

Implementation and deployment: We implemented and deployed Kaleidoscope to Blue Waters' Cray Sonexion-1600 [4] petascale distributed storage system. This distributed storage system uses Lustre file system, [5] a file system that is used by more than 70% of the top 100 supercomputers [62]. The key result of this work is summarized below.

High accuracy: Kaleidoscope is highly accurate for failure localization and failure detection. We use two years of Blue Water's production data with 843 known production issues flagged by the system operators. Kaleidoscope can localize failures for 99.3% of the cases and infer the correct failure mode and cause of the failure in 95.8% of the cases. Kaleidoscope can distinguish between failures caused by component failures and resource overload failures. To our surprise, Kaleidoscope finds failures that are not previously identified by the human operator as well.

Low false positive rate: A monitoring system that generates a large number of false positives can distract the operators from focusing on the real issue. Kaleidoscope produces fewer false positives than the state-of-the-art monitoring technique, Net Bouncer [63], by 100 times.

[4] Sonexion 1600 Hardware Architecture (cray.com): https://pubs.cray.com/bundle/Sonexion_1600_Field_Installation_Guide_1.5.0_H-6124/page/Sonexion_1600_Hardware_Architecture.html

[5] Lustre: https://www.lustre.org/

Low resource overheads: Kaleidoscope and the Store Pings monitoring probs are non-intrusive to the normal system operations. The measured overhead of Kaleidoscope is less than 0.01% of the system's peak I/O throughput.

## 6.2 Kaleidoscope: Related Work

There are a wide variety of failure localization techniques for HPC [63]–[70]. Kaleidoscope goes beyond failure localization as it also infers the failure mode and the root cause of the failures for choosing the correct mitigation strategy. Kaleidoscope, to the best of our knowledge, is the first method to offer such capabilities, and it differentiates itself from the crowd in the following aspects: 1) Existing works based on supervised ML methods [69], [70] are data-hungry, while Kaleidoscope improves data efficiency by using domain-driven PGM model. 2) Previous works support either failure detection and localization (e.g., [63], [67]) or failure diagnosis (e.g., [71]), while Kaleidoscope supports all three while handling the noise in the data due to non-deterministic routing and system behaviors. 3) Kaleidoscope recognizes failures in advance before the clients are affected, and uses active probing called Store Pings to actively monitor the health states of the components. Kaleidoscope places the Store Pings strategically to reduce active monitoring overhead while providing comprehensive coverages. Overall, Kaleidoscope is a low-overhead, high-accuracy, high-coverage failure recognition framework and it is the first method that can distinguish between reliability failures and resource contention failures.

## 6.3 Kaleidoscope: Background

1) Blue Waters storage system: The Cray Sonexion storage system of the Blue Waters supercomputer is a large-scale distributed storage system that can support up to petabytes of computing throughputs. The system architecture is described in detail in Chapter 5, so we omit the details here. In summary, the storage system consists of 6 management servers (MGS), 6 metadata servers (MDS, or MS in the later sections), and 420 object storage servers (OSS, or data server (DS), in later sections). The storage servers are connected by a storage network called Infiniteband network, and the compute nodes (clients) connect to the storage system via 582 I/O load balancers, also known as I/O nodes. The Cray Sonexion storage system uses a total of 17280 HDDs as data storage devices to store the file data, and these disks are configured in RAID to improve I/O performance and redundancy. The disks are referred to as object storage devices or OSDs. Each DS is connected to one or more clusters of OSDs, and servers themselves form high-availability (HA) pairs.

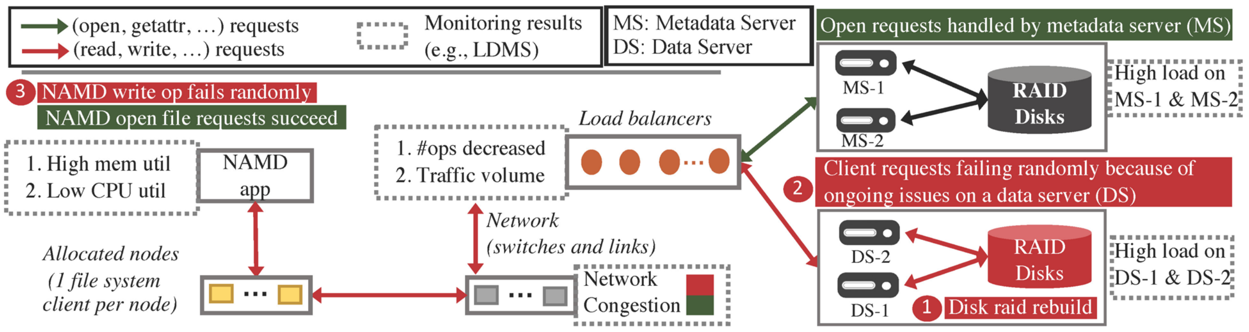

(open, getattr, ...) requests
(read, write, ...) requests
Monitoring results (e.g., LDMS)
MS: Metadata Server
DS: Data Server
Open requests handled by metadata server (MS)
3 NAMD write op fails randomly
NAMD open file requests succeed
1. High mem util
2. Low CPU util
NAMD app
Load balancers
1. #ops decreased
2. Traffic volume
MS-1
MS-2
RAID Disks
High load on MS-1 & MS-2
2 Client requests failing randomly because of ongoing issues on a data server (DS)
Network (switches and links)
Allocated nodes (1 file system client per node)
Network Congestion
DS-2
DS-1
RAID Disks
High load on DS-1 & DS-2
1 Disk raid rebuild

*Figure 22: I/O Failure Propagation with Limited Views on the Actual Issue*

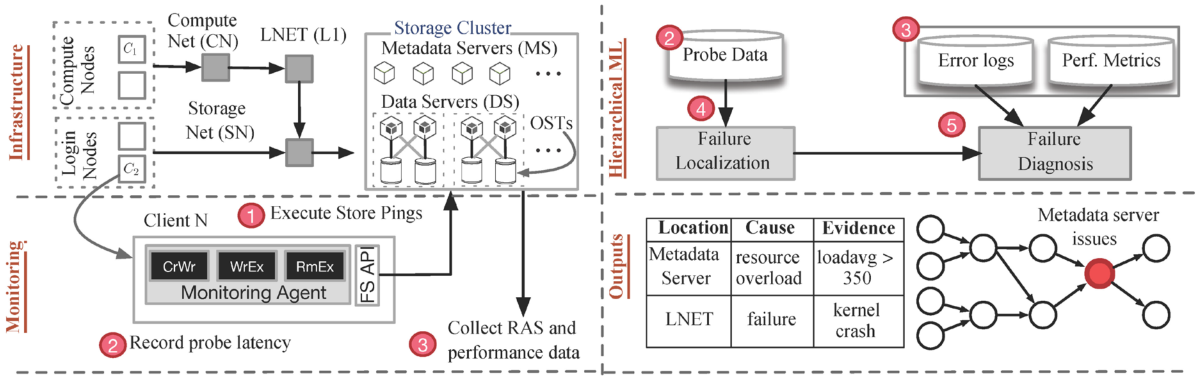
2) Failure case study: Figure 22 shows a typical real-world failure scenario. Such failure is hard to pin-point and analyze using the telemetry data (performance measurement of the system) alone because the telemetry data provide a partial view of the problem. On the storage server side (data server), the telemetry data indicate high loads and degraded performance of the storage system, while the telemetry data from other healthy data servers do not reflect these high loads. From the running applications on the clien side (a compute node), the file operations will likely be slow or failing, hinting at a possible problem in the file system client, in the network, or in the storage system. None of these data point directly to the source of the issue, so they are not sufficient for component-level failure localization and diagnosis. The precise reasons for failure are buried deeply in the system logs, as the subsequent analysis of the system logs reviews a disk failure with its effects propagating throughout the system. The failure disk triggers a RAID rebuild and drastically decreases the available bandwidth. Such reduction in bandwidth leads to increasing system loads and finally triggers server failures due to the loads. All these are revealed from the system logs rather than the telemetry data.

3) Challenges: The failure case presented above highlights several challenges. Telemetry data collected in a large-scale system is inherently heterogeneous because each subsystem uses different monitoring techniques and generates a different set of performance measurements, thereby presenting only a partial view of the system states.

Therefore, the first challenge is to jointly analyze these data. The second challenge is failure data collection and labeling because failures are rare events, and most of the failures are transient and one-off. It is hard to both identify a failure and subsequently label it if the failure is a new kind of failure. The third challenge is the presence of noise in the measurements. The measurements are noisy because of propagation delays, randomness in dynamic routing, and the existence of short-term failures that affect the measurements. Finally, the fourth challenge is the timeliness of the analysis. The analysis must provide comprehensive system coverage while ensuring the critical failures are detected sufficiently before they impact the system. The challenges make it hard to analyze these telemetry data automatically using rule-based algorithms because there are endless rules to encode. The only option left for the system operators is to dig through terabytes of data manually and hope to understand the cause. Due to this, it is hard to perform timely mitigation to alleviate the negative impact of the failures on the system. It is often the case that failure causes are found only after the failure has happened.

## 6.4 Kaleidoscope: List of Components

Figure 23 below shows a list of components of the Kaleidoscope. The top-left shows a simplified view of the Kaleidoscope. The lower-left shows the telemetry data collection facility. The top-right shows the overall model for failure localization and diagnosis. The lower-right shows the interpretable result of the Kaleidoscope.



| Location | Cause | Evidence |
|---|---|---|
| Metadata Server | resource overload | loadavg > 350 |
| LNET | failure | kernel crash |

*Figure 23: Kaleidoscope Framework Design Overview*

With all these components, Kaleidoscope addresses each of the four challenges presented in Section 6.3 as follows. To address the first challenge, Kaleidoscope fuses heterogenous telemetry data that captures both the storage system's and client's views and analyzes them jointly. To address the second challenge, Kaleidoscope uses PGM, a class of unsupervised ML methods, to remove the need for explicit data labeling. To address the third challenge, Kaleidoscope models system uncertainty and compensates for noise using hierarchical probabilistic ML models that leverage domain knowledge to model the measurement noise and failure propagation effects. Finally, to address the fourth challenge, Kaleidoscope uses low-cost monitoring and ML method to reduce monitoring and failure localization/diagnosis overheads and can perform failure localization and diagnosis in real time.

## 6.5 Kaleidoscope: Monitoring and Telemetry Data Collection

1) End-to-end Probing Monitors: Kaleidoscope uses Store Pings, a set of monitoring probes that monitor the latency over paths. Here, a path is a route from the

client to the storage server that allows the client to ultimately access the desired file. Store Pings provide health information of a route based on a basic principle: for the file operation to succeed end-to-end (complete the operation within the time limit), all components on that route must be healthy. Store Pings not only record the total latency of the request but also allow the request to be routed through a specific set of load balancers for more deterministic behavior. Assuming the topology of the HPC is fixed in the short term since the I/O pings generated by the Store Pings are of fixed size, the request completion time is likely fixed. Therefore, for any Store Pings request that takes longer than one second, we declare the request as timeout because 99% of the Store Pings request return within one second.

The Store Pings request from a client to a file uniquely identifies the data server and the possible path it can take since the metadata server has metadata information for the file pinged from that request. The ping also reduces the number of possible load balancers and routes that the request can take since an OSD can only connect up to four load balancers. Though not 100% deterministic (as that will require changing the load balancing algorithm specifically to support Store Pings), the numbers of possible routes are significantly reduced. This also makes it tractable to model the route uncertainty using PGM.

Store Pings improve observability by using a wide range of existing file operations. There are three types of Store Pings: CrWr, WrEx, and RmEx, which stand for create-then-write, write-to-existing-file, and remove-existing-file. CrWr and RmEx test the functionality of the metadata server (MS) in file creation and removal respectively, while WrEx tests the functionality of both the MS and the data server (DS). Issued from the client, these three types of Store Pings test all the components necessary to complete file requests, including the client itself, the load balancers, the client-to-storage-system network, the storage system network, and the storage servers.

To achieve maximum coverage while incurring minimum monitoring overheads, Store Pings are placed strategically in the system. The placement of Store Pings is guided by the sufficient identifiability condition described in [72], [73], so we omit the details here. In general, Store Pings are placed on clients that have different system stacks, live on different networks, and perform different services. Specifically, Store Pings are placed on all service nodes that provide scheduling (64 services nodes), importing and exporting of bulk data (25 I/E nodes), and user login nodes (user entry point, 4 nodes). We monitor these nodes because they provide critical services for the end-users.

At each monitoring time step (every minute), Store Pings are executed from all 4 login nodes, 1 out of 64 services nodes, and 1 out of 25 I/E nodes. Store Pings ping every OSD, DS, and MS, which results in 72 CrWr and 72 RmEx pings (from 6 clients to 6 MS and 6 OSDs), and 5184 WrEx pings (from 6 clients to 6 DS and 432 OSDs).

2) Component logs: Apart from Store Pings, Kaleidoscope uses a comprehensive monitoring system to collect performance measurements and RAS logs for all system components. These data are collected by either the Light-weight Distributed Metric Service (LDMS) [74] or the Integrated System Console (ISC) [75]. These logs provide more fine-grained information on component health states.

## 6.6 Kaleidoscope: Methodology

Kaleidoscope uses hierarchical machine learning models for real-time failure localization and failure diagnosis given the failures exist. The hierarchical machine learning model consists of two parts: (i) failure localization model for identifying failure components, and (ii) failure diagnosis model for identifying failure modes and the root cause of the failure. The failure localization model is primarily developed by Saurabh Jha with his permission to reprint the work so we include it here for completeness. The failure diagnosis model is developed as a joint effort by Shengkun Cui and Saurabh Jha.

1) Failure localization model: The failure localization model identifies the components that are failed or overloaded and result in I/O failures. Store Pings telemetry data is used for failure localization. However, Store Pings data are inherently noisy, like other telemetry data, due to asynchronous data collection, dynamic routing, load balancing, and the presence of transient failures, and provide only a partial view of the system (latency measured for a single route). We need to consider the uncertainty in

these telemetry data and fuse the partial views to get an accurate estimation of the system's health.

Kaleidoscope uses a probabilistic graphical model (PGM) in the form of a factor graph (FG) [76] to model uncertainties as probabilistic random variables, and fuse the telemetry data from all Store Pings monitors together by jointly analyzing them. The PGM models the relationships between the random variables (components in the system) using a graphical structure, with nodes being the random variables, and the edges between random variables representing the statistical relationships between these random variables. Such a structure allows the PGM to model complex conditional independence relationships between random variables—variables that are conditionally independent are not connected—based on domain-specific knowledge. We use the PGM formulation because we believe that PGM is able to reduce the measurement errors by jointly analyzing groups of Store Pings measurements. Kaleidoscope uses a specific form of PGM called the factor graph (FG) that provides the most flexible form of formalism to model the system using factor functions. In our FG model, the failure states of the individual components on a path are graph nodes and treated as hidden variables, and the corresponding path availability is also a graph node and it is an observed variable. The relationship between the hidden and the observed variables on a path is captured using factor functions—a set of statistical functions that model the statistical relationship between related variables.

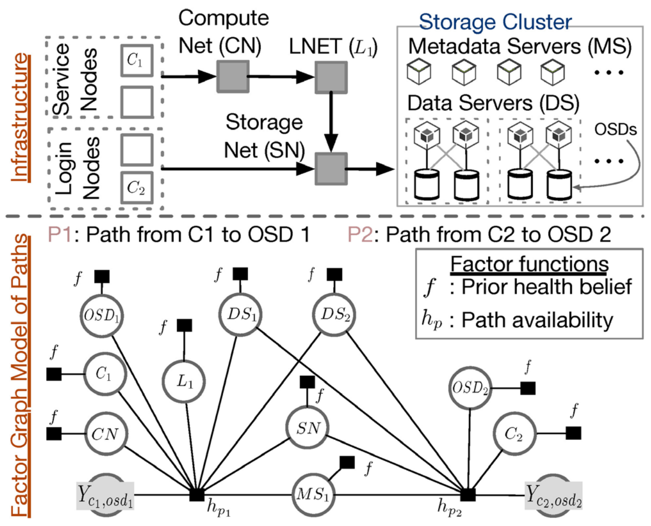


*Figure 24: Factor Graph used in the Failure Localization Module*

Formulating the problem: The formalism of the problem using the FG is shown in Figure 24. The health and hence the failure state of a component $i$ on a path is at time $t$ denoted as $X_i$ (omitting the $t$ for simplicity), and its value captures the probability that the component can serve the I/O request successfully. Without prior measurements, we model $X_i \sim \text{Beta}(\alpha, \beta)$ from Beta distribution as its prior. At each time step, $\alpha$, $\beta$ are updated based on the measurements (observations). At each time step, the Store Pings monitors generate reachability measurements between client $C_i$ and OSD $OSD_j$, and this measurement, due to uncertainty, can be denoted as a random variable $Y_{<C_i,OSD_j>}$. Since Store Pings issues multiple independent pings, $Y_{<C_i,OSD_j>}$ is derived from a binomial distribution to capture the overall success rate of the requests. Here, $Y_{<C_i,OSD_j>} \sim Binomial\left(A_{<C_i,OSD_j>}, N\right)$ in which $A_{<C_i,OSD_j>}$ denotes the probability of reachability from the client to that OSD, and $N$ is the total number of Store Pings issued from $C_i$ to $OSD_j$ in the time interval $(t-1, t]$. Note that the FG's inference time interval is independent of the time interval of the Store Pings, and FG uses some accumulation of measurement from the Store Pings. To model $A_{<C_i,OSD_j>}$ we used the topology of the system and the domain knowledge of the statistical relationships between the telemetry data and the component's health to calculate $A_{<C_i,OSD_j>}$. These statistical relationships are based on the understanding of the system topology and routes. Here, $A_{<C_i,OSD_j>}$ solely depends on the product of the probabilities of being healthy of all individual components (assuming that components are independent of each other), in other words, all components on the requested route need to be healthy for the request to serve successfully: $A_{<C_i,OSD_j>} = \prod_{\mathrm{i} \in \mathrm{Path}(\mathrm{C_i},\mathrm{OSD_j})} X_i$.

Recall that the routing is dynamic due to load balancers and HA server pairs so routing is non-deterministic. We incorporate such non-determinism into the model by using a series-parallel network formalism. For example the Store Pings request eventually lands on the OSD that contains the file. The OSD is connected by two DSs, named DS-1 and DS-2. The OSD is not available if both of the DSs are down; otherwise, the healthy DS can serve the request. The $R_{OSD_i}$, the probability of a file access request completing successfully from a destinated load balancer to OSD, is $R_{OSD_i} = \left(1 - (1 - X_{DS-1}) \cdot (1 - X_{DS-2})\right) \cdot X_{OSD_i}$, where the $X_{DS-1}, X_{DS-2}$, and $X_{OSD_i}$ denote the probability that DS-1, DS-2, and the i-th OSD are healthy, respectively. The formula $(1 - X_{DS-1})$ denotes the probability that DS-1 has failed, and similarly for $(1 - X_{DS-2})$, so the product of those is the probability that both DS-1 and DS-2 have failed, so $\left(1-(1-X_{DS-1})\cdot(1-X_{DS-2})\right)$ is the probability that at least one of the DS-1 and DS-2 is healthy, so the request can be served. Finally, $\left(1-(1-X_{DS-1})\cdot(1-X_{DS-2})\right)$ multiplied with the probability of OSD being healthy ($X_{OSD_i}$) is the probability of $R_{OSD_i}$ being healthy. We use a similar way to model other redundancy and uncertainty measurements such as multiple load balancers. Ultimately, $A_{<C_i,OSD_j>} = X_{C_i}\cdot X_{L_i}\cdot X_{CN_i}\cdot X_{SN_i}\cdot X_{MS_i}\cdot R_{OSD_i}$, with $C_i, L_i, CN_i, SN_i, MS_i$ being the client, LNET, compute network, storage network, and metadata server (MS) respectively. Here, $X_{L_i}$ should be similar to $R_{OSD_i}$ when modeling redundancy, so we omit here for brevity. Apart from modeling the system with spatial relationships (topology), Kaleidoscope also models temporal relationships (evolution over time) between previous and current time steps by using the estimated parameters from the previous time step as priors for estimations in the current time step, assuming that the failures propagate through the system gradually.

The above relationships are encoded in the factor functions of the FG. As shown in Figure 24, the components are FG nodes with the unshaded ones being the hidden variables that we wish to infer, and the shaded ones being the observed variable from the measurements. In Figure 24, the black boxes are factor functions, with the singular function $f$ being the prior beliefs of the components' health, and $h_{p_i}$ are path functions following the above multivariate binomial formulation. The inference of the FG is to calculate the expected health of the components (hidden variables) given the set of observations $E(X_1, X_2, X_3, \ldots | Y_1, Y_2, Y_3 \ldots)$, which requires the knowledge of both the observation and the priors of the components' health. The observations are the measurements of the successful Store Pings during a specific interval, and the priors $f$ follow the Beta distribution, with the individual parameters $\alpha, \beta$ being estimates from the inference result in the previous time step (for $t = 0$, we set $f = 0.5$ as there is no prior information). The role of inference task is to update these parameters for the priors based on the telemetry data observations. Kaleidoscope solves the inference problem using the Monte Carlo Markov Chain algorithm [77], which estimates the expected health of the components given the observations by generating a large number of samples from the model while tweaking the parameters to minimize the disparities between the predicted observations from the sampled distribution as well as the real observations. Our model declares the component to be failed when the confidence in inferencing is over 75%. The FG is implemented using PyMC3 [78, p. 3] with observations collected over 5 min intervals.

2) Failure diagnosis model: The primary goal of the failure diagnosis model is to identify the failure mode as well as the root cause of the failure given the set of telemetry data including performance metrics and RAS logs. The failure diagnosis model answers the conditional question "Given a component has failed, which modality of the telemetry data best explains the failure?", and the condition "given a component has failed" is obtained from the failure localization model. This question is very different

from answering the unconditional version of the question “Which modality of the telemetry data is anomalous across all components?” The non-conditional version of the question requires the model to identify the failures from the logs first and then identify the failure modes and root cause, which is quite challenging as mentioned because of the partial observability and the noise present in the telemetry data. For example, it is common that a server has higher loads than the rest of the server, but we cannot flag the server based on just high loads as the file request can still be served within the time limit. However, if the failure localization model, after combining all data, identifies that the server has failed, then high loads might be a root cause of the failure.

The failure detection model answers the conditional question by statistically comparing the measurements of the failed component and the healthy component by using an unsupervised anomaly detection model that selects the measurement conveying the failure modes. If there is a reliability failure, the failure detection model points to the failure logs, and if there is a resource overload failure, the model points to the performance metric that indicates that failure.

Diagnosing reliability failures: Reliability failures are identified by analyzing the system logs. We work with vendors and national labs to create a library of regular-expression patterns from five years of logs. These regular expressions cover a wide range of logs under failures, and we use these regular expressions to filter out error logs from normal logs. Kaleidoscope filters RAS logs of the components by using these regular expressions, and the error logs of each failed component are compared to the error logs of the healthy components: $\delta = L_{failed} - \cup_{i \in \{healthy\}} L_i$; if the difference is not an empty set, then the failure is caused by component failures.

Diagnosing resource overload failures: The difference between resource overload failures and reliability failures is that in the presence of the former there might be no component failures. Therefore, Kaleidoscope focuses on performance indicators such as load average, disk await-time, latency, and network performance counters to identify resource overload failures. Kaleidoscope identifies resource overload failures by using the local outlier factor (LOF) algorithm [61] to compare the performances of similar storage components. The LOF is based on the local density of the data points in which the density is given by the distance of the K nearest neighbors. By calculating the LOF for all components of a similar type, we can identify the failure components as outliers. If a failed component has the same LOF score as the others on the same performance metric, there is no reason to believe that the component is marked failed due to resource overload and vice versa. LOF is a reasonable choice because storage components within a homogeneous group could have different modes of operation that are not indications of anomalies. We found that some groups of data servers have a low load average of 10, but some other groups have a high load average of 64, and both groups operate as intended. Kaleidoscope declares that a failed component is due to resource overload if its $LOF_r$ is 1.5 times greater than the max LOF of the value of the healthy components.

Finally, there is no explicit training involved in constructing the failure diagnosis module as we use unsupervised methods with domain-specific knowledge. The method does require sufficient knowledge of the system to come up with a set of regular expressions for filtering the error logs.

## 6.7 Kaleidoscope: Evaluation and Discussion

We evaluate Kaleidoscope by using two years of Store Pings monitoring data collected from the Cray Sonexion storage system starting from 2018 as well as live forensics on Cray Sonexion for over three months. Over the two-year span, Cray Sonexion operators have identified and resolved 843 production issues that led to critical failures of the storage system, and each issue has a report on the cause of the problem. These issues are used as ground truth failures in the Kaleidoscope evaluation as true positives, and others as true negatives. We also randomly select 100 issues identified by kaleidoscope to quantify the number of false positives.

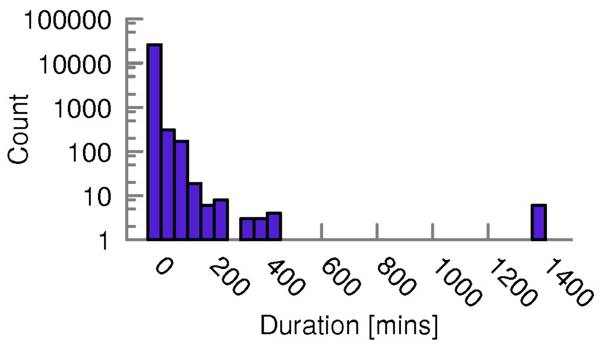


*Figure 25: Duration of System Outage Distribution*

1) Effectiveness: Kaleidoscope identifies 26596 I/O failure events with 25427 resource overload failures and 1169 reliability failures. The large number of failures identified by Kaleidoscope in comparison to the failure identified by the operators is due to transient failures. Transient failures are short-term low-impact failures that can be masked by the system's redundancy or recovered with the system's recovery mechanisms so they do not lead to production issues. Cray's operators use the following two policies to identify important I/O failures for manual investigation: 1) The failure alarms are discarded if there are subsequent recovery events that recover the system. 2) Resource overload failures are often transient and an investigation is initiated only if the failures persist over 30 minutes, shown as the tail in Figure 25. These two policies are applied to the failures found by Kaleidoscope and the total number of failures is reduced to 1525.

Kaleidoscope can localize the failures for 837 out of 843 cases, with an accuracy of 99.3%, and those 6 undetected failures did not impact the system but were flagged only for maintenance purposes. The additional 668 detected failures are not false positives as many of the failures point to a real underlying problem of the system after close inspection. Those 668 failures are unidentified failures mainly due to lack of monitoring so they were ignored by the operators.

In those 843 production issues, 346 issues were reliability failures and 497 were resource overload failures. Kaleidoscope is able to correctly identify 340 reliability failures and their causes, and 468 resource overload failures and their causes, which leads to the diagnostic accuracy of 98.3% and 94.2% respectively. Among the 668 failures detected by the failure localization module, Kaleidoscope additionally recognizes 22 reliability-related failures and 558 resource overload-related failures. After a manual inspection of 100 out of these 558 resource overload-related failures, we found that many of them were true and they were not false positives.

It is hard to precisely quantify the false positive rate because the failures flagged by Kaleidoscope could indeed represent a true failure but were ignored by the system operators. Therefore, 100 failures identified by Kaleidoscope, with half of them flagged as reliability failures and half of them flagged as resource overload failures, are selected for analysis. The result indicates that Kaleidoscope localizes all 100 failures correctly but incorrectly identifies 4 out of these 100 failures.

*Table 5: Kaleidoscope Comparing with NetBouncer in Failure Localization. Using six months of production data.*

| | **True Positive** | **False Negative** | **Alarms** |
|---|---|---|---|
| Kaleidoscope | 184 | 2 | 4892 |
| NetBouncer | 110 | 76 | 116072 |

2) Baseline comparison: Kaleidoscope is compared with NetBouncer [63], a state-of-the-art failure localization method for HPC, using six months of previous production data. The result of the comparison is shown in Table 5. We can see that compared to Kaleidoscope, NetBouncer generates many more alarms and misses more cases. This could be due to the inability of NetBouncer to compensate for uncertainty in request

routing and evolution of the component states. Kaleidoscope tackles both issues much better by using hierarchical machine learning techniques, as discussed in Section 6.6.

3) Monitoring overheads: Another important perspective of any monitoring and failure recognition technique is the monitoring overheads. Kaleidoscope aims to have low monitoring overheads and is non-intrusive to the normal operations of the system. The monitoring overheads of Kaleidoscope are benchmarked using the IOR benchmark, [6] a tool to test the Lustre file system performance. During our evaluation, there are 4320 compute nodes running IOR. The results indicate that when running Store Pings and Kaleidoscope with 100 Store Pings monitors running at 30 sec intervals, the peak performance is 97.58% of the peak performance of the system without Store Pings and Kaleidoscope. Note that in a production environment the user applications can hardly stress the storage system to its peak performance. We also evaluate the time difference between subsequent launches of Store Pings. The results show that over 98% of Store Pings are launched within 3 sec, and all of them are launched within 10 sec.

[6] GitHub - LLNL/ior: Parallel filesystem I/O benchmark: https://github.com/LLNL/ior

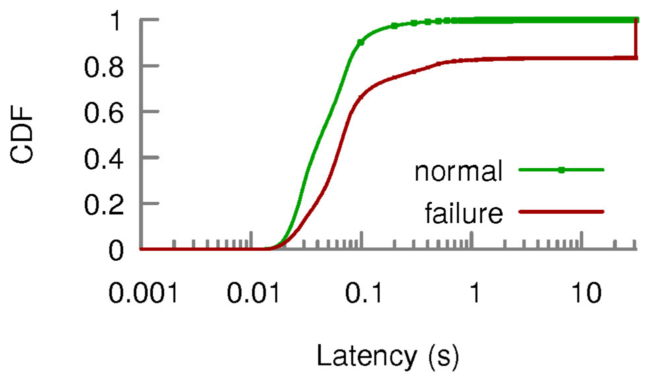


*Figure 26: End-to-End Latency of Store Pings, green without fault, red with faults*

4) System operation experiences: The deployment of Kaleidoscope helps the system operators to identify production issues as well as enabling fine-grained failure characterization of the storage system.

Kaleidoscope finds that in most cases reliability failures lead to system performance degradations which subsequently lead to I/O request failures. However, only a small percentage <1% of reliability failures lead to system-wide outage (SWO). This is because most of these component failures are masked by the system's redundancies. For example, at the disk-level, RAID arrays act as redundancies to protect the system from disk failures, and at the server-level, HA-pairs protects the system against server failures. Nevertheless, most of these failures trigger system recoveries that divert some bandwidth from the user applications, and hence the performance loss. Figure 26 shows that the average I/O request completion time increased by 52.7% under failures with the 99$^{th}$ completion time at 31 sec (request time out).

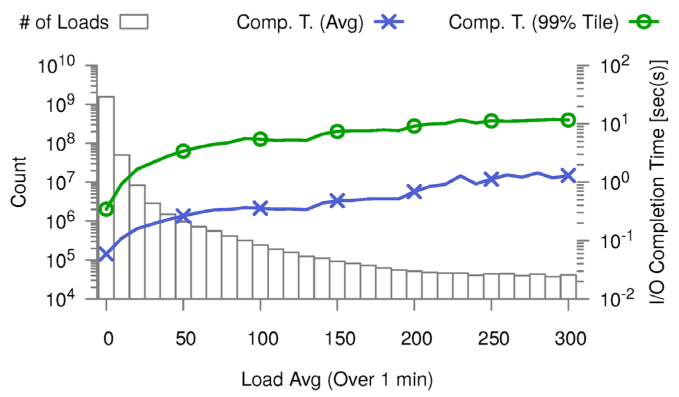


*Figure 27: Load Average Duration and Its Affects I/O Completion Time*

On the other hand, Kaleidoscope reveals that resource overhead failures due to excessive I/O by the application or high load on the server frequently lead to I/O request failures. Excessive I/O causes high loads on the server and the storage devices and therefore slows down the entire system. As shown in Figure 27, generally, excessive I/O abuses are short and within 10 sec. However, the relatively long tail in the figure indicates that there exist some applications with excessive I/O requests as long as 4 h. During that incident, the system load average index increases from 60 to 350. Moreover, system high load is another reason for resource overload failures. The I/O request completion time is strongly correlated with the load on the storage servers. As shown in Figure 27, when the average server load indices increase, so does the average I/O request completion time and the 99th percentile I/O request completion time. A severely high load can cause the I/O request completion time to increase by more than 10 times. The high load can be caused by either excessive I/O or self-recovery/rebuild of the storage server.

Finally, Kaleidoscope can help the system operators identify one-off failures that were unseen before because the failure localization model localizes failures based on telemetry data rather than historical events (training data in the supervised models). On average four new failure modes are found per month. One such case is when the system experienced a partial failure in the LNET node, causing the bandwidth available from the clients to that DS to decrease by 25%. Such a partial failure was novel because though the LNET node had failed, it continued to generate heartbeats as if alive. Kaleidoscope can quickly localize such failure by telemetry data and point to the failed LNET node.

5) Generalization: A valuable monitoring technique needs to be generalizable to different systems. Kaleidoscope is not developed specifically for the Cray Sonexion storage system or Lustre file system. Kaleidoscope uses PGM for failure localization, and RAS logs and performance metrics for failure diagnosis, and none of these is tied to a specific system. Performance metrics and RAS logs are available on every HPC system and assuming the topology of the system is known, developers can derive the PGM structure from the system topology. One unique feature provided by Kaleidoscope is Store Pings, which provides the ability to periodically ping from a client to a storage device by issuing I/O requests. Other systems have similar supports. For example, Ceph distributed file system [79] uses the CRUSH [80] algorithm to compute file location. We can use CRUSH to get the file mapping rules, and strategically place the files on every

storage device and ping the files by issuing I/O requests from clients. Therefore, though Store Pings monitors are proprietary to Kaleidoscope, constructing a similar monitoring facility in other HPC systems is not intractable.

## 6.8 Kaleidoscope: Conclusion

This chapter presents Kaleidoscope, an ML-based failure localization and diagnosis framework for HPC systems. Kaleidoscope uses PGM and unsupervised methods to localize and diagnose system failures in real time, and can effectively distinguish reliability failures from resource overload failures for the best mitigation plan. In addition, Kaleidoscope uses Store Ping monitors to provide high coverage while incurring low monitoring overheads. We evaluate Kaleidoscope using two years of production data from the Blue Waters supercomputer's Cray Sonexion storage system and show that Kaleidoscope not only recognizes real production failures with high accuracy but also helps system operators identify unseen failures. Finally, we argue that Kaleidoscope is generalizable to other HPC systems because it does not require specific system architectures or software facilities to achieve its full functionality. We believe that Kaleidoscope will be a powerful tool in failure recognition for HPC systems that can help the system operators to better monitoring and maintaining their system.

# Chapter 7: Related Work

This chapter discusses work related to autonomous driving system (ADS) security and reliability assessments, as well as high-performance computing (HPC) system failure monitoring, localization, and diagnosis.

## 7.1 ADS Security Assessment

Security assessment on ADS has mainly focused on the deep learning-based machine learning models as they are notoriously prone to adversarial attacks, so one line of work is to assess the ADS ability to withstand adversarial attacks—attacks that aim to "fool" the deep learning-based model to produce incorrect predictions. Carlini and Wagner [81] reveal that modern convolutional neural network (CNN)-based image classifiers are prone to carefully designed adversarial noise. Such noise is not obvious to human eyes but could lead to distribution shifts when added to the input data and cause misclassifications, for example, classifying a panda as a truck. As shown in [27]–[31], the authors target the CNN-based obstacle detector used by different sensors, which are the core of the ADS perception pipelines. Works such as [27] target CNN-based obstacle detector for camera and show successful attacks on stop sign by painting certain textures onto the stop sign. Similarly, [31] shows similar but more robust attacks. In addition, [28] targets CNN-based obstacle detector for LiDAR by placing carefully designed physical adversarial obstacles on the road that are not detectable by the obstacle detector using LiDAR data, causing potential accidents. Finally, [30] targets end-to-end camera-based ADS by painting black stripes on the road, causing the ADS to make the wrong steering decisions. This line of work considers only the impacts of adversarial attacks on the targeted detector to cause misdetection, without considering the impact on the AV safety. Specifically, existing works do not consider the importance of timing or defensive capabilities brought by the spatial and temporal redundancy in the ADS. Work such as [32] considers the temporal redundancy from the obstacle tracker (Kalman filter) and provides a more advanced attack method to attack the entire multiple-obstacle-tracking (MOT) pipeline, but it too fails to consider the timing of the attack which is proven to be utterly critical in our work RoboTack.

Another line of work seeks to gain access to ADS. Various works show attacks that gain access to the ADS by hacking vehicle-to-vehicle (V2V) channels [82], [83], vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) channels [84], OTA software updates [85], ECUs [70], CAN bus [86], infotainment systems [87], and physical access to the physical vehicle [82]. All these works show that it is possible to hack into the ADS and gain critical knowledge, which makes the RoboTack attack entirely possible.

## 7.2 ADS Reliability Assessment

ADS is a complex system consisting of artificial intelligence and machine learning-based algorithms and heterogeneous hardware to meet the performance and compute requirements. ADS, like any other complex systems, is also prone to reliability failures such as transient and permanent faults. Modern ADS uses both hardware-based and software-based techniques to improve system redundancy.

One such technique is duplicating the software and hardware completely and running the two stacks independently, and comparing the outputs of the duplicated system bit-by-bit. This technique, while effective at identifying both transient and permanent faults, is very costly.

Other techniques improve system resiliency at the hardware level, as introduced in Section 4.2. Hardware techniques improve system resiliency by introducing redundancy on instruction execution such as lockstep execution [43], bit error detection using parities such as ECC memory and parity checks [40], [42], or other techniques such as thread redundancy [44], [45]. The improvement of hardware resiliency to faults reduced the need to duplicate the entire system, so that only part of the hardware required modification. However, hardware redundancy incurs higher manufacturing costs due to increased complexity and reduces performance due to increased redundancy overheads.

Fault detection and correction can also be included at the software level without dedicated hardware support. These techniques include error detection and correction by modifying algorithms. For example, [46] proposed a fault-tolerant matrix multiplication algorithm, and [47] improves the resilience of convolutions by algorithm-based error detection. Other categories include software runtime monitoring [49] and diverse execution of the process on duplicated processors [50]. The software-based assessment methods do not require specific hardware architecture, which further reduces the manufacturing cost. However, software-based techniques often target a specific type of software in the ADS system; for example, [46], [47] target matrix arithmetic in the DNNs, while other techniques require specific software supports (such as compiler support) or redesign of the software. Therefore, unlike DiverseAV which can assess the entire ADS, existing software-based methods can only assess part of the ADS software, or require a substantial redesign of the software.

## 7.3 HPC Failure Localization and Diagnosis

There is a wide range of attempts on fault detection and localization in the HPC system [63]–[70]. Many of these techniques focus on fault detection and localization in specific areas of the distributed system. For example, [63], [64] use active probing techniques for network latency measurement and anomaly detection. Works like [65] use Lasso regression and hypothesis testing for timely fault localization for the virtual disk. Works like [66] propose a user-friendly end-to-end I/O monitoring tool with a focus on automatic I/O anomaly localization by analyzing historical data. In addition, [67] focuses on the detection of fail-slow faults—in which the hardware or software functions but with degraded performance—using time-out signals. There are also works on failure diagnosis. Moreover, [71] diagnoses system faults by comparing the abnormal system logs with healthy system logs and points to the most influential factor for the developers. Few of these works can simultaneously detect, localize, and diagnose system faults like the Kaleidoscope. Finally, works like [70] that simultaneously perform fault detection and localization using deep learning-based methods ([70] uses recurrent neural network for system log processing) are often data-hungry and require collecting and labeling a large amount of training data. Kaleidoscope avoids this problem and improves data efficiency by using domain-knowledge-driven graphical model and unsupervised learning methods.

# Chapter 8: Conclusion and Future Work

## 8.1 Conclusion

In this thesis, we present three AI-driven assessment methods in the context of two applications: autonomous driving system (ADS) and high-performance computing (HPC) system. Each method addresses a unique set of challenges. We propose RoboTack, an end-to-end attack framework targeting ADS in the form of smart malware. RoboTack considers both the spatial and temporal redundancy of the ADS and uses ML methods to learn the vulnerability within the ADS perception and sensor fusion system, achieving a high attack success rate. RoboTack, with the capability to automatically identify ADS security vulnerabilities in critical scenarios, can help the developers to quickly identify and fix these vulnerabilities via simulations.

Moreover, to address reliability assessment in ADS, we propose DiverseAV, a high-accuracy and high-coverage fault detection framework that solves the challenge of low-cost reliability assessment in ADS. DiverseAV achieves high accuracy and high coverage (over 85% precision and recall in hardware permanent fault detection, and 99% precision and recall in detecting software failures) by leveraging natural temporal diversity in sensor data and vehicle state-conditioned thresholds—a first of its kind. In addition, DiverseAV does not rely on proprietary hardware or hardware duplication, but requires only a duplication of the software, reducing manufacturing cost. Furthermore, DiverseAV does not require redesigning the ADS software, which makes it highly generalizable to various ADS.

Finally, we introduce Kaleidoscope, a low-overhead, high-accuracy, and high-coverage framework that simultaneously performs fault detection, localization, and diagnosis on HPC systems. The two key challenges that Kaleidoscope solves are 1) data-efficient failure localization and diagnosis and 2) distinguishing between reliability failures and resource overload failures. The latter is particularly important because, given that the impacts of the two types of failures on the end-users are similar, knowing the root cause of the failure promotes the correct mitigation strategy (reliability failures—restart, resource overload failures—throttling). Kaleidoscope achieves these solutions by combining domain knowledge of the system with probabilistic graphical models and unsupervised machine learning methods. Moreover, Kaleidoscope is not tailored for a particular HPC system, but rather is a general framework for fault localization, detection, and diagnosis on HPC systems.

## 8.2 Future Work

The work presented in this thesis provides valuable future research directions in improving the resiliency of complex ADS and HPC systems. Our work on RoboTack reveals that the widely adopted CNN-Kalman-filter-based MOT (multiple-obstacle-tracking) pipelines and sensor fusion models in modular-based ADS are susceptible to malicious attacks at critical times, contrary to the conclusions of previous works. This vulnerability is caused by the inability of the pipeline to distinguish between legitmate noisy measurements and malicious perturbed measurements, causing the pipeline to erroneously update the state estimation using perturbed measurements. One direction of future work that may reduce such vulnerability is to develop tracking and sensor fusion methods that can identify malicious measurements from legitimate measurements. Another direction is to introduce an uncertainty measurement to obstacle detectors so that the detectors report uncertainty envelopes of their detections. This could improve the perception pipeline in defending against malicious attacks that cause high uncertainty in obstacle detections. Though learning uncertainties of the model and the data have been proposed in several works [88], [89], they have not yet been applied to the control feedback loop of the ADS.

In addition, we introduce DiverseAV for low-cost transient and permanent fault detections in ADS, but we have not discussed the mitigation plan once a critical fault is detected. A line of research on DiverseAV is to propose fine-grained strategies for fault mitigation in the ADS settings, as currently the most common mitigation process is limited to disengagement and handing controls back to the human driver. A detailed mitigation procedure will require better understanding of the detected faults (fault diagnosis), and we argue that the failure diagonosis model in Kaleidoscope can be applied to ADS as well.

Lastly, we assert that Kaleidoscope is not limited to HPC systems, but rather can be used for other complex systems as well. The intuition behind Kaleidoscope is to leverage domain-driven knowledge to more efficiently localize and diagnose system faults. The formulation of Kaleidoscope is generally applicable to a wide range of complex systems, because the complex system, software or hardware, has topology and causal relationships between components. Therefore, a line of future work can focus on fault detection and localization on other cyber-physical systems, such as unmanned aerial vehicles (UAVs) and complex control systems in critical infrastructures (e.g., smart grids), using Kaleidoscope's formalism.

Overall, the emergence of AI-driven assessment methods not only enables timely security and reliability assessment of complex systems but also opens up new directions for system resiliency research. We anticipate that more AI-driven methods will be adopted in future systems, and with the aid of these methods, the future generation of systems will be more powerful, more intelligent, and more resilient.

# References

[1] T.S., “Why Uber’s self-driving car killed a pedestrian,” *The Economist*, May 29, 2018.

[2] M. Snir, W. D. Gropp, and P. Kogge, “Exascale research: Preparing for the post-Moore era,” Univesity of Illinois white paper, 2011.

[3] D. Shapiro, “Introducing Xavier, the NVIDIA AI supercomputer for the future of autonomous transportation,” NVIDIA blog post, Sep. 28, 2016.

[4] R. Moore-Colyer, “Nvidia Xavier supercomputer aims to turn cars into AIs on wheels,” Silicon.co.uk, Jan. 05, 2017.

[5] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G. S. Sachdev, “Compute solution for Tesla’s full self-driving computer,” *IEEE Micro*, vol. 40, no. 2, pp. 25–35, 2020, doi: 10.1109/MM.2020.2975764.

[6] S. Alvarez. (2018, Jun.). Research group demos why Tesla autopilot could crash into a stationary vehicle. [Online]. Available: https://www.teslarati.com/tesla-research-group-autopilot-crash-demo/.

[7] S. Jha, S. Cui, S. S. Banerjee, T. Xu, J. Enos, M. Showerman, Z. T. Kalbarczyk, and R. K. Iyer, “Live forensics for HPC systems: A case study on distributed storage systems,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–16, doi: 10.1109/SC41405.2020.00069.

[8] Baidu Apollo team. (2019). Apollo: Open source autonomous driving. [Online]. Available: https://github.com/ApolloAuto/apollo

[9] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, “Autoware on board: Enabling autonomous vehicles with embedded systems,” in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, 2018, pp. 287–296.

[10] G. Welch and G. Bishop, *An Introduction to the Kalman Filter*. University of North Carolina, 1995.

[11] J. Redmon and A. Farhadi, “YOLOv3: An incremental improvement,” *ArXiv Prepr. ArXiv180402767*, 2018.

[12] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in *Proceedings of Advances in Neural Information Processing Systems*, 2015, pp. 91–99.

[13] N. Garnett, R. Cohen, T. Pe’er, R. Lahav, and D. Levi, “3D-LaneNet: End-to-end 3D multiple lane detection,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 2921–2930, doi: 10.1109/ICCV.2019.00301.

[14] F. Pizzati and F. Garcia, “Enhanced free space detection in multiple lanes based on single CNN with scene identification,” *2019 IEEE Intell. Veh. Symp. IV*, Jun. 2019, doi: 10.1109/ivs.2019.8814181. [Online]. Available: http://dx.doi.org/10.1109/IVS.2019.8814181

[15] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese, “Social LSTM: Human trajectory prediction in crowded spaces,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 961–971, doi: 10.1109/CVPR.2016.110.

[16] F. Altché and A. de L. Fortelle, “An LSTM network for highway trajectory prediction,” in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, 2017, pp. 353–359, doi: 10.1109/ITSC.2017.8317913.

[17] S. LaValle, “Rapidly-exploring random trees: A new tool for path planning,” *Annu. Res. Rep.*, 1998.

[18] K. J. Aström and T. Hägglund, *PID Controllers: Theory, Design, and Tuning vol. 2*. Instrument Society of America Research Triangle Park, NC, 1995.

[19] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, “End to end learning for self-driving cars,” *ArXiv Prepr. ArXiv160407316,* 2016.

[20] D. Chen, B. Zhou, V. Koltun, and P. Krähenbühl, “Learning by cheating,” in *Conference on Robot Learning*, *PMLR 100,* 2020, pp. 66-75.

[21] F. Codevilla, M. Müller, A. López, V. Koltun, and A. Dosovitskiy, “End-to-end driving via conditional imitation learning,” in *Proceedings of International Conf. on Robotics and Automation (ICRA)*, 2018.

[22] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An open urban driving simulator,” in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[23] A. Tampuu, M. Semikin, N. Muhammad, D. Fishman, and T. Matiisen, "A survey of end-to-end driving: Architectures and training methods," in *IEEE Transactions on Neural Networks and Learning Systems*, Dec. 2020.

[24] S. Jha, S. Cui, S. Banerjee, J. Cyriac, T. Tsai, Z. Kalbarczyk, and R. K. Iyer, "ML-driven malware that targets AV safety," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 113–124, doi: 10.1109/DSN48063.2020.00030.

[25] B. C. Csáji, "Approximation with artificial neural networks," MSc thesis, Eötvös Loránd University, Hungary, 2001.

[26] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim, "LGSVL simulator: A high fidelity simulator for autonomous driving," *ArXiv Prepr. ArXiv200503778*, 2020.

[27] J. Lu, H. Sibai, and E. Fabry, "Adversarial examples that fool detectors," *ArXiv Prepr. ArXiv171202494*, 2017.

[28] Y. Cao, C. Xiao, D. Yang, J. Fang, R. Yang, M. Liu, and B. Li, "Adversarial objects against LiDAR-based autonomous driving systems," *ArXiv Prepr. ArXiv190705418*, 2019.

[29] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Artificial Intelligence Safety and Security*, R. V. Yampolskiy, ed., CRC Press, 2016, pp. 99-112.

[30] A. Boloor, K. Garimella, X. He, C. Gill, Y. Vorobeychik, and X. Zhang, "Attacking vision-based perception in end-to-end autonomous driving models," *ArXiv Prepr. ArXiv191001907*, 2019.

[31] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning models," *ArXiv Prepr. ArXiv170708945*, 2017.

[32] Y. Jia, Y. Lu, J. Shen, Q. A. Chen, H. Chen, Z. Zhong, and T. Wei, "Fooling detection alone is not enough: Adversarial attack against multiple object tracking," in *International Conference on Learning Representations*, 2020.

[33] S. Jha, S. Banerjee, T. Tsai, S. K. S. 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 *Proceedings of 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 112–124, doi: 10.1109/DSN.2019.00025.

[34] K.-T. Cho and K. G. Shin, "Fingerprinting electronic control units for vehicle intrusion detection," in *Proceedings of 25th USENIX Security Symposium*, 2016, pp. 911–927.

[35] J. Lu, H. Sibai, E. Fabry, and D. Forsyth, "No need to worry about adversarial examples in object detection in autonomous vehicles," *ArXiv Prepr. ArXiv170703501*, 2017.

[36] D. Rezvani, "Hacking automotive ethernet cameras," *Argus Cyber Security*, Nov. 2018. [Online]. Available: https://argus-sec.com/hacking-automotive-ethernet-cameras/

[37] O. Pfeiffer, *Implementing Scalable CAN Security with CANcrypt*, Embedded Systems Academy Inc., 2017.

[38] NEOUSYS. Nuvo-6108GC GPU computing platform. [Online]. Available: https://www.neousys-tech.com/en/product/application/gpu-computing/nuvo-6108gc-gpu-computing

[39] G. L. Hicks, L. D. Howe Jr, and F. A. Zurla Jr, "Instruction retry mechanism for a data processing system," U.S. Patent no. 4,044,337. 23 Aug. 1977.

[40] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Microelectron. Div.*, vol. 11, 1997.

[41] R. Nathan and D. J. Sorin, "Argus-G: Comprehensive, low-cost error detection for GPGPU cores," *IEEE Comput. Archit. Lett.*, vol. 14, no. 1, pp. 13–16, 2015, doi: 10.1109/LCA.2014.2298391.

[42] R. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[43] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The Arm triple core lock-step (TCLS) processor," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, Jun. 2019, doi: 10.1145/3323917.

[44] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, “Detailed design and evaluation of redundant multi-threading alternatives,” in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002, pp. 99–110, doi: 10.1109/ISCA.2002.1003566.

[45] E. Rotenberg, “AR-SMT: A microarchitectural approach to fault tolerance in microprocessors,” in *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, 1999, pp. 84–91, doi: 10.1109/FTCS.1999.781037.

[46] K.-H. Huang and J. A. Abraham, “Algorithm-based fault tolerance for matrix operations,” *IEEE Trans. Comput.*, vol. 100, no. 6, pp. 518–528, 1984.

[47] S. K. S. Hari, M. Sullivan, T. Tsai, and S. W. Keckler, “Making convolutions resilient via algorithm-based error detection techniques,” *IEEE Trans. on Dep. and Secur. Comp.*, pp. 1–1, 2021, doi: 10.1109/TDSC.2021.3063083.

[48] L. A. Clarke and D. S. Rosenblum, “A historical perspective on runtime assertion checking in software development,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 25–37, 2006.

[49] N. Delgado, A. Q. Gates, and S. Roach, “A taxonomy and catalog of runtime software-fault monitoring tools,” *IEEE Trans. Softw. Eng.*, vol. 30, no. 12, pp. 859–872, 2004.

[50] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella, “Software-only based diverse redundancy for ASIL-D automotive applications on embedded HPC platforms,” in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2020, pp. 1–4.

[51] A. Avizienis, “The N-version approach to fault-tolerant software,” *IEEE Trans. Softw. Eng.*, no. 12, pp. 1491–1501, 1985.

[52] J. P. J. Kelly, T. I. McVittie, and W. I. Yamamoto, “Implementing design diversity to achieve fault tolerance,” *IEEE Softw.*, vol. 8, no. 4, pp. 61–71, 1991.

[53] P. E. Ammann and J. C. Knight, “Data diversity: An approach to software fault tolerance,” *IEEE Trans. Comput.*, vol. 37, no. 4, pp. 418–425, 1988.

[54] National Highway Traffic Safety Administration (NHTSA), “Pre-crash scenario typology for crash avoidance research,” DOT HS 810 767, 2007.

[55] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, “Quantifying the accuracy of high-level fault injection techniques for hardware faults,” in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014, pp. 375–382.

[56] PinFI. [Online]. Available: https://github.com/DependableSystemsLab/pinfi

[57] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, “NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 372–383.

[58] NVBitFI. [Online]. Available: https://github.com/NVlabs/nvbitfi

[59] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, “PyTorchFI: A runtime perturbation tool for DNNs,” in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2020, pp. 25–31, doi: 10.1109/DSN-W50199.2020.00014.

[60] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for autonomous driving? The Kitti vision benchmark suite,” in *2012 IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, pp. 3354–3361.

[61] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, “LOF: Identifying density-based local outliers,” in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD’00)*, Dallas, Texas, USA, 2000.

[62] S. Oral and F. Baetke, LUSTRE community BOF: Lustre in HPC and emerging data markets: Roadmap, features and challenges. [Online]. Available: https://sc18.supercomputing.org/proceedings/bof/bof_pages/bof176.html

[63] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, “NetBouncer: Active device and link failure localization in data center networks,” in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI’19)*, Boston, MA, USA, 2019.

[64] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, “Pingmesh: A large-scale system for data center network latency measurement and analysis,” in *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM’15)*, London, United Kingdom, 2015.

[65] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, and M. Chintalapati, A. Krishnamurthy, and T. Anderson, “Deepview: Virtual disk failure diagnosis and pattern detection for Azure,” in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI’18)*, Renton, WA, USA, 2018.

[66] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, “End-to-end I/O monitoring on a leading supercomputer,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019, pp. 379–394.

[67] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi, “IASO: A fail-slow detection and mitigation framework for distributed storage services,” in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC’19)*, Renton, WA, 2019.

[68] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, “Pinpoint: Problem determination in large, dynamic internet services,” in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 595–604.

[69] A. Das, F. Mueller, P. Hargrove, E. Roman, and S. Baden, “Doomsday: Predicting which node will fail when on supercomputers,” in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 108–121.

[70] M. Du, F. Li, G. Zheng, and V. Srikumar, “Deeplog: Anomaly detection and diagnosis from system logs through deep learning,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1285–1298.

[71] K. Nagaraj, C. Killian, and J. Neville, “Structured comparative analysis of systems logs to diagnose performance problems,” Presented at 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), San Jose, CA, 2012, pp. 353–366.

[72] L. Ma, T. He, A. Swami, D. Towsley, and K. K. Leung, “Network capability in localizing node failures via end-to-end path measurements,” *IEEEACM Trans. Netw.*, vol. 25, no. 1, pp. 434–450, 2016.

[73] L. Ma, T. He, A. Swami, D. Towsley, K. K. Leung, and J. Lowe, “Node failure localization via network tomography,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*, New York, NY, USA, 2014, pp. 195–208, doi: 10.1145/2663716.2663723.

[74] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, et al. “The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications,” in *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[75] M. Butler. Blue Waters super system. [Online]. Available: https://www.globusworld.org/files/2010/02/SuperSystem-BW-.pdf

[76] D. Koller, N. Friedman, and F. Bach, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[77] R. M. Neal, *Probabilistic Inference Using Markov Chain Monte Carlo Methods*, University of Toronto, 1993.

[78] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, “Probabilistic programming in Python using PyMC3,” *PeerJ Comput. Sci.*, vol. 2, p. e55, Apr. 2016, doi: 10.7717/peerj-cs.55.

[79] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI’06)*, Seattle, WA, 2016.

[80] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, “CRUSH: Controlled, scalable, decentralized placement of replicated data,” in *SC’06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, pp. 31–31.

[81] N. Carlini and D. Wagner, “Towards evaluating the robustness of neural networks,” *arXiv preprint arXiv:1608.04644*, 2017.

[82] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. “Experimental security analysis of a modern automobile,” in *Proceedings of 2010 IEEE Symposium on Security and Privacy*, 2010, pp. 447–462.

[83] Z. Winkelman, M. Buenaventura, J. M. Anderson, N. M. Beyene, P. Katkar, and G. C. Baumann, *When Autonomous Vehicles Are Hacked, Who Is Liable?* Santa Monica, CA: RAND Corporation, 2019.

[84] I. A. Sumra, I. Ahmad, H. Hasbullah and J. bin Ab Manan, “Classes of attacks in VANET,” in *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*, Riyadh, Saudi Arabia, 2011, pp. 1-5, doi: 10.1109/SIECPC.2011.5876939.

[85] A. Sampath, H. Dai, H. Zheng, and B. Y. Zhao, “Multi-channel jamming attacks using cognitive radios,” in *Proceedings of 2007 16th International Conference on Computer Communications and Networks*, 2007, pp. 352–357.

[86] E. Yağdereli, C. Gemci, and A. Z. Aktaş, “A study on cyber-security of autonomous and unmanned vehicles,” *J. Def. Model. Simul.*, vol. 12, no. 4, pp. 369–381, 2015.

[87] T. Lin and L. Chen. (2019). Common attacks against car infotainment systems. [Online]. Available: https://events19.linuxfoundation.org/wp-content/uploads/2018/07/ALS19-Common-Attacks-Against-Car-Infotainment-Systems.pdf

[88] A. Kendall and Y. Gal, “What uncertainties do we need in Bayesian deep learning for computer vision?” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2017, pp. 5580–5590.

[89] J. Choi, D. Chun, H. Kim, and H. Lee, “Gaussian YOLOv3: An accurate and fast object detector using localization uncertainty for autonomous driving,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 502–511, doi: 10.1109/ICCV.2019.00059.