ASSESSING DEPENDABILITY OF EMERGENT LARGE-SCALE AUTONOMOUS SYSTEMS IN THE WILD

BY

SAURABH JHA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Ravishankar K. Iyer, Chair
Professor Wen-mei W. Hwu
Professor William T. Kramer
Professor Tianyin Xu
Professor Steve Keckler, The University of Texas at Austin and NVIDIA

ABSTRACT

Emergent computer systems in transportation, healthcare, and enterprise systems are increasingly adopting data-driven techniques using machine learning and artificial intelligence to automate their operation, management, and control. Their widespread use in mission-critical services that involve humans means that it is of paramount importance to provide an ever-increasing level of runtime system dependability. Dependability is a cross-cutting issue spanning the system stack, including hardware, software, and algorithms that compose the system. In addition to existing challenges, such as issues of failures, load balancing, and scalability, a significant challenge arises from the fact that these systems must make decisions in the presence of uncertainties stemming from the system (e.g., transient failures), environment/data (e.g., out-of-training distribution data), and computational models (e.g., inadequate training). An erroneous decision by the system, if not detected, will lead to silent failures and degradation that will propagate to all layers of the system, ultimately leading to catastrophic outcomes. Therefore, data-driven automation combined with the ever-increasing scale and complexity has exposed these systems to emerging failures, attacks, and performance degradation modes that are difficult to deal with using existing techniques in a dynamically evolving, multi-tenant environment. The phenomenon is exemplified by several newsworthy headlines, such as an Uber self-driving car colliding with and killing a pedestrian.

This thesis develops novel data-driven methods and techniques for assuring dependability by (i) understanding the fundamental challenges to achieving system dependability that emerge due to the use of data-driven automation techniques, (ii) rigorously validating the system, including its runtime operational characteristics, and (iii) developing runtime monitoring techniques to detect, identify, and isolate events that threaten system dependability. The methods proposed in this thesis have been demonstrated on significant and broad user-inspired cases of societal importance with significantly different dependability requirements: (i) autonomous vehicles (AVs), and (ii) large-scale high-performance computing (HPC) systems.

Dedicated to my brother and parents.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisers, Professor Ravishankar K. Iyer and Professor Zbigniew Kalbarczyk, for their tremendous support and invaluable advice during the various stages of the development of this dissertation. They always encouraged me to pursue exciting new directions, pushed me to step beyond the established boundaries, and gave me the courage to put forward controversial ideas. It is their confidence that kept me motivated and determined to complete this thesis. Each has been a friend, a mentor, and a role model and taught me the essential skills of becoming an independent researcher and an effective communicator.

This work would not have been possible without the help and support of dissertation committee members Professor William T. Kramer (Director of the Blue Waters Project and NCSA's (National Center for Supercomputing Applications) @Scale Programs), Dr. Steve Keckler (VP of Architecture Research at NVIDIA and Adjunct Professor of Computer Science at the University of Texas at Austin), Professor Wen-mei W. Hwu (Senior Distinguished Research Scientist at NVIDIA and Emeritus Professor at the University of Illinois at Urbana-Champaign (UIUC)), and Professor Tianyin Xu (in the Computer Science Department at the UIUC). Over the years, the committee members thoroughly critiqued and shaped my work and imparted both the research and soft skills without which this work would not have been possible.

I would also like to thank all my collaborators across universities, industries, and national laboratories who significantly invested time and effort in mentoring and critiquing my work. Specially, Jim Brandt and Ann Gentile at Sandia National Laboratories and Larry Kaplan and Mark Dalton at Cray collaborated on high-performance computing; Timothy Tsai, Siva Hari, Mike Sullivan, and Steve Keckler at NVIDIA on hardware resiliency and AV safety; Mike Showerman, Jeremy Enos and Gregory Bauer at NCSA collaborated on application and system monitoring; Lidong Zhou and Shobha Balakrishnan from Microsoft Research and Hari Ramaswamy, Amos Omokpo, Karthick Rajamani on cloud resiliency and availability. I would also like to thank Professor Saurabh Bagchi and Rakesh Kumar at Purdue University, Bentolhoda Jafary and Associate Professor Lance Fiondella at the University of Massachusetts Dartmouth, and Assistant Professor Guanpeng Li at the University of Iowa for their contributions to this dissertation. My sincere thanks to other collaborators: Eric Roman, Taylor Groves, Annette Greiner, and Steve Leak at NERSC, Jonathan Petit at Qualcomm, Brett Bode and many others at NCSA,

Amanda Bonnie and Mike Mason at Lawerence Livermore National Lab, who directly or indirectly contributed to this dissertation.

Over the years, I worked with several undergraduate and masters students at UIUC, who trusted in me to provide guidance and partnership and contributed significantly to my dissertation work. This experience significantly boosted my confidence. My special thanks to Shengkun Cui (now at NVIDIA), Archit Patke (now Ph.D. student at UIUC), Benjamin Lim (now a masters student at CMU), James Cyriac (now at Microsoft), Yiran Li, Lavin Devnani (now at NVIDIA), Fei Deng (now at Verizon Media), Sharon Tang (now at Microsoft), and Yan Miao (now a masters student at UIUC).

The research described in this dissertation could not have been possible without the help of many wonderful colleagues in the DEPEND group and at the Coordinated Science Laboratory (CSL) at the University of Illinois. I especially thank Arjun Athreya, Subho Bannerjee, Catello Di Martino (Lelio), Valerio Formicola, Yogathessan Varathrajah, Homa Alemzadeh, Krishnakant Saboo, Chang Hu, Yuming Wu, Pooja Malik, Zach Stephens, Key-whan Chung, Zak Estrada, Phuong Cao, Cuong Pham, Hui Lin, Daniel Chen, Ted Hong, Archit Patke, Haoran Qui, and Haotian Chen for the many insightful research discussions. Arjun, Subho, and Lelio went above and beyond in providing both friendship and mentorship during these years. I am deeply grateful to Heidi Leerkamp and Kathy Atchley for their kind assistance with many administrative tasks and continuous guidance. I am also grateful to Jenny Applequist, Fran Rigberg, and Kathy Atchley for their patience in reviewing my papers and this dissertation. I also thank the CSL communications team, including August Schiess, Allie Arp, and Kim Gudeman, for their continuous efforts to showcase my research outcomes. I am also thankful to Maggie Chappell, Kara MacGregor, and Viveka Kudaligama in the Computer Science Department, who went above and beyond to help me out on several occasions.

I would not have been able to complete this dissertation without the constant support and compassion of my friends, who always believed in me. I want to thank my friends in India (and abroad), Ankita Sharma, Ayesha Shamsi, Anmol Ghosh, Gaurav Gupta, Pranav Verma, Vasant Hedge, Ananya Patra, Megha Panda, and Manisha Jha, and in the USA, Arjun Athreya, Sumit Mudgal, Subho Banerjee, Uttam Thakore, Atul Bohra, Varun Badrinath, Tejaswi Agarwal, Ahbinandan Patni, and Tarique Siddiqui.

I would like to dedicate this thesis to my brother Nigam Jha, who believed in me and encouraged me to pursue my dreams. Without him, I would not have decided to study computer science and later pursue Ph.D. He inspired me daily through his actions, hard work, and patience dedicated to pursuing his lifelong passion. Finally, I would like to thank my parents, Maula Nand Jha and Vidya Jha, for their love and support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Emergent computer systems in transportation, healthcare, and enterprise systems are increasingly adopting data-driven techniques using machine learning and artificial intelligence to automate their operation, management, and control. Their widespread use in mission-critical services that involve humans means that it is of paramount importance to provide an ever-increasing level of runtime system dependability[1]. In addition to existing challenges, such as issues of failures, load balancing, and scalability, a significant challenge arises from the fact that these systems must make decisions in the presence of uncertainties stemming from the system (e.g., transient failures), environment/data (e.g., out-of-training distribution data), and computational models (e.g., inadequate training) (see Fig. 1.1). An erroneous decision taken by the system, if not detected, will lead to silent failures and degradation that will propagate to all layers of the system, ultimately leading to catastrophic outcomes. Therefore, data-driven automation combined with the ever-increasing scale and complexity has exposed these systems to emerging failures, attacks, and performance degradation modes that are difficult to deal with using existing techniques in a dynamically evolving, multi-tenant environment, as exemplified by several newsworthy headlines [2–5].

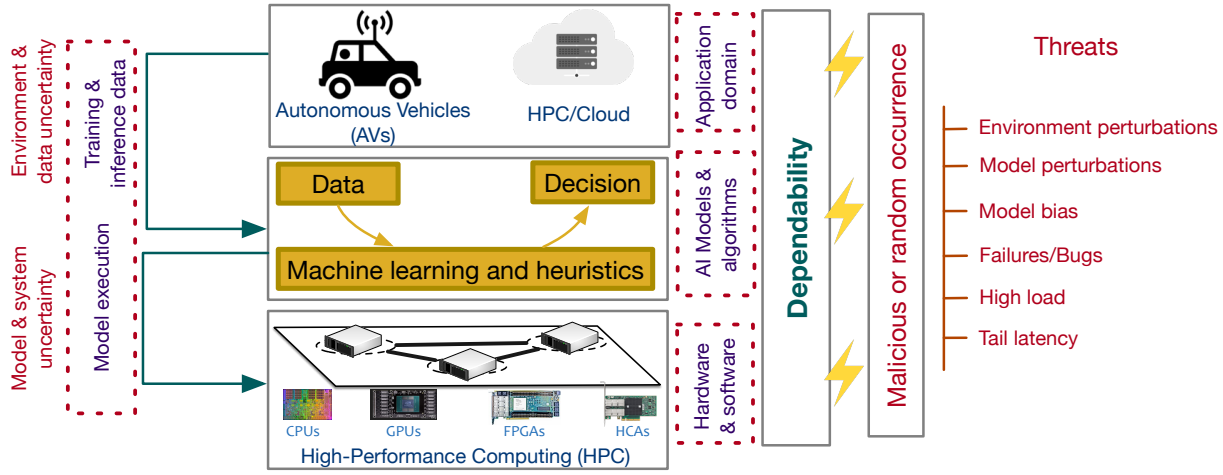


Figure 1.1: Threats & Challenges in achieving dependability in mission-critical systems.

[1]Avizienis et al. define dependability, in [1], as an integrating concept that encompasses the following attributes: availability, reliability, safety, integrity, and maintainability. We extend this definition to include performance and latency, as these attributes have significant impact on safety of a control-driven system.

1.1 GOALS AND USE-CASES

Dependability is a cross-cutting issue spanning the system stack, including hardware, software, and algorithms that compose the system. The goal of this thesis is to assure dependability by (i) understanding the fundamental challenges to achieving system dependability that emerge due to the use of data-driven automation techniques, (ii) rigorously validating the system, including its runtime operational characteristics, and (iii) developing runtime monitoring techniques to detect, identify, and isolate events that threaten system dependability.

This thesis uses significant and broad user-inspired cases of societal importance with significantly different dependability requirements: (i) autonomous vehicles (AVs), and (ii) large-scale high-performance computing (HPC) systems. We use AVs as a use case because (i) they are highly heterogeneous and fully autonomous systems and (ii) they showcase the difficulties in assessing ML-driven systems [6]. We use HPCs because (i) they are fundamental to scientific computing and (ii) they serve as a bedrock for enabling latency-sensitive and computationally demanding autonomous applications, such as self-driving cars. Autonomous systems such as self-driving cars generate petabytes of data and use that data for training and inference in real time. The ability to process such large amounts of data while meeting latency deadlines can only be enabled via high-performance computing.

1.2 RESEARCH CHALLENGES AND SUMMARY OF CONTRIBUTIONS

This section first provides a summary of technical contributions across different application domains and then describes the innovations in each application domain.

Empirical assessment: An empirical assessment of field datasets from real-world production systems enables (i) discovery of failure modes and operational characteristics encountered in the field; (ii) quantification of failure statistics, the relative contribution of each failure mode, and failure propagation paths; and (iii) development and identification of assertions to guide verification, validation, and runtime monitoring. However, using field-failure datasets is challenging, as the internal details of the system may not be available or interpretable (e.g., in the case of DNNs). Moreover, these datasets are inherently noisy and incomplete, forcing the user to make assumptions about the system that may not be true. This thesis addresses these challenge by designing data analysis techniques that overlay field-failure datasets with an abstract representation of the control system, enabling us to pose factual and counterfactual questions (use causal reasoning)

to examine the observed safety hazards. Using empirical techniques, we have uncovered insights and results that highlight the need to fundamentally redesign and reinvent dependability techniques. For example, our empirical results show that adversarial attacks [7] and random transient faults [8] have almost negligible impact on the vehicle's safety due to temporal and spatial resiliency. Therefore, the attacker or robust dependability techniques must focus on end-to-end system dependability. Recent techniques that focus only on ML/AI dependability significantly mischaracterize the problem as these techniques suffer from over-simplification of equating ML models to autonomous systems.

Design-time assessment and validation: The evolutionary, context-sensitive behavior of such systems can cause unexpected emergent behavior or unforeseen interactions that were not necessarily envisioned at the architectural stage of the system design. Moreover, as these systems are expected to work without human oversight, these data-driven, mission-critical systems must be rigorously validated and tested in the pre-deployment phase to ensure their dependability in the field. However, it is difficult to assess and validate such systems for several reasons, including: (i) Although conducting real-world validation tests in the field is valuable, it is not viable to test mission-critical systems with real workloads. For example, testing approaches like chaos engineering with real user traffic [9] would be extremely dangerous for humans in the case of self-driving cars and could lead to significant loss of money and value in case of bank transactions, in both cases posing significant ethical risks. (ii) Enumerating over fault/attack space and inputs, which is combinatorially large, is infeasible. This thesis addresses those challenges in two ways: (i) Testing the end-to-end system properties, with both hardware and software in the loop, in a controlled environment. Such a testing approach avoids real-world human subjects but is realistic at the same time. (ii) Modeling the problem of identifying critical adverse events (such as high load, faults, rare and invalid inputs, attacks) that threaten dependability as a machine-learning problem under the causal framework. Not only do these techniques help validate the system, they also enable curation of standard benchmarks for comparison and testing of systems in a scalable way. We are the first to use causal and counterfactual analysis, especially do-calculus, for testing large-scale autonomous systems.

Runtime assessment: Due to monetary cost and design difficulties, it is infeasible and often impossible to identify all adverse events that threaten the system properties; thereby, leaving the system with a significant number of defects. Hence, these systems are bound to experience failures leading to catastrophic outcomes in the field, as exemplified by several newsworthy headlines [2–5]. In this thesis, we address issues related

to both (i) the software and hardware and (ii) algorithms. Traditionally, system designers have relied on designing assertions (e.g., heartbeats) and redundancy techniques (e.g., checksum and voting) for runtime monitoring and handling of adverse events. However, such techniques do not scale well for large-scale systems, as there are hundreds of thousands of components. Moreover, heartbeat-based techniques are misleading because they hide partial failures and are subject to system noise [10]. Similarly, designing test- (or probe)-based assertion techniques for every single component on a case-by-case basis is infeasible. Redundancy techniques, such as duplication, do not remove defects present in the models and heuristics used by these systems. N-version programming or ensemble of models is appealing for these systems. However, these solutions are costly and, in some cases, exacerbate the problem due to the high uncertainty and low accuracy of these models [11]. To address those challenges, this thesis develops end-to-end online testing techniques that use causal models not only to evaluate the dependability of the system but also to assess the risk associated with taking action so as to proactively reduce risk and avoid catastrophes.

Next, we describe the detailed contributions in each of the two use-case application domains.

1.3 AUTONOMOUS VEHICLES

Autonomous vehicles (AVs), such as self-driving cars and unmanned aerial vehicles, are complex systems that use artificial intelligence (AI) and machine learning (ML) to integrate mechanical, electronic, and computing technologies to make real-time driving decisions. AI enables AVs to navigate complex environments while maintaining a *safety envelope* [12, 13] that is continuously measured and quantified by onboard sensors (e.g., camera, LiDAR, RADAR) [14–16].

Industry-grade AVs are equipped with detection and mitigation techniques to handle dependability challenges; however, a significant number of failures silently escape detection. Such silent failures, if not dealt with, lead to erratic driving behavior and safety hazards, thereby reducing the trust we place on them, as exemplified by several headline-making AV crashes [3, 2]. Moreover, an adversary can masquerade an attack as a silent failure by intelligently perturbing the environment or models to evade detection and successfully cause a safety hazard [17]. Hence, there is a compelling need for a comprehensive assessment of AV technology to identify and handle those silent failures.

Characterizing production dataset. We analyzed the field-failure dataset on disen-

gagements and accidents of self-driving cars (Chapter 2). There are several challenges in analyzing field dataset on self-driving cars across different car manufacturers: (i) manufacturers have not disclosed the architectures of their autonomous vehicles, and (ii) these datasets are inherently noisy and incomplete. To address these challenges, we developed LogDriver [6], which uses a system theoretic process analysis (STPA)-based causal model to construct a hypothesized control structure of a model self-driving car based on technical documentation [18–22]. We used LogDriver to identify multidimensional causes of AV disengagements/accidents from field datasets and show several results.

The California Department of Motor Vehicles (CA DMV) mandates that all manufacturers testing AVs on public roads file annual reports detailing *disengagements* (a failure that causes the control of the vehicle to switch from the software to the human driver) and *accidents* (an actual collision with other vehicles, pedestrians, or property) [23]. We analyzed field data collected over a 26-month period from September 2014 to November 2016 (part of the DMV's 2016 and 2017 data releases), containing data from 12 AV manufacturers for 144 vehicles that drove a cumulative $1,116,605$ autonomous miles. Across all manufacturers, we observed a total of $5,328$ disengagements, 42 of which led to accidents. The analysis shows the following: (i) For the same number of miles driven, for the manufacturers that reported accidents, human-driven non-AVs were $15-4000\times$ less likely than AV's to have an accident. (ii) 64% of disengagements were the result of problems in, or untimely decisions made by, the machine learning system. (iii) In terms of reliability per mission, AVs are $4.22\times$ worse than airplanes, and $2.5\times$ better than surgical robots. (iv) Trend analysis of disengagements and accidents per mile reveal that while individual components of AV technology (e.g., vision systems, control systems) may have matured, entire AV systems are still in a "burn-in" phase. The analysis presented in this thesis shows a distinct improvement in the performance of AVs over time. However, it also demonstrates the need for continued improvement in the dependability of this technology.

Scaling validation techniques. A vital issue for self-driving cars is that of rigorously demonstrating and validating their safety. The evolutionary, context-sensitive behavior of autonomous systems can cause unexpected emergent behavior or unforeseen interactions that were not necessarily envisioned at the architectural stage of the system design. The causes of "unexpected" behaviors include unforeseen interactions between autonomy and vehicle, various notions of failure and hazard scenarios, faults (design and physical), and security threats. However, it is difficult to assess and validate such systems for several reasons, including: (i) Although conducting real-world validation tests in the field is valuable, it is not viable to test mission-critical systems with real-workloads.

For example, testing approaches like chaos engineering with real user traffic [9] would be exteremely dangerous for humans in case of self-driving cars and could lead to significant loss money and value in case of bank transactions, thereby, posing significant ethical risks. (ii) Enumerating over fault/attack space and inputs, which is combinatorially large, is infeasible.

We demonstrated the use of causality-driven models, which we implemented using probabilistic graph models, to assess the safety of autonomous vehicles (AVs) with respect to reliability and security vulnerabilities (Chapter 3). We developed the *Bayesian Fault Injector (BFI)* [24], an intelligent resiliency assessment tool that can identify situations and faults that will likely lead to violations of safety and resiliency requirements. The BFI relies on (i) a fault injection (FI) engine, (ii) an ML-based fault selection engine, and (iii) safety models (e.g., collision avoidance in AVs) and reliability models (e.g., of the ability to tolerate up to k failures). Fault injection is the process of deliberately introducing faults in the system by corrupting values of variables or corrupting software or hardware components to analyze the system behavior in the presence of faults. It is difficult and often unnecessary to simulate all possible faults in the system. We developed an ML-based fault selection engine for causal and counterfactual reasoning about the system state in terms of safety and reliability under a fault scenario. The functional relationship among the system software (represented by a control-flow graph), the system state (safety and reliability), and safety/reliability is modeled using Temporal Bayesian Networks (TBNs), a probabilistic graphical model (PGM)-based ML approach, thus bridging the gap between model-based techniques [25] and data-driven techniques [9]. TBNs are trained using system execution traces. TBNs significantly reduce the need to gather training data (system traces), which in turn reduces the computational overhead and training time.

BFI is highly effective and scalable (1600× faster than traditional methods) in finding bugs (5 unique bugs) and failure patterns (500 unique failure patterns) that can lead fatal vehicle collisions.

Exposing security vulnerabilities. The above-mentioned techniques are also used by the attackers to identify runtime vulnerabilities in the system. However, a challenge for an attacker is to hide the footprint and evade detection. One approach to evade detection is to masquerade the attack as a naturally occurring random fault in the system. However, it is challenging to masquerade attacks as faults in AVs because of the inbuilt compensation in the system and environment. For example, state tracking algorithms such as Kalman and particle filters tolerate random noise. Similarly, an attack launched on an AV on an empty road will not result in a collision. Moreover, the intrusion detection system can detect the attack if the attacker maintains its presence for too long in the

system.

The goal of our work is to identify the steps that an adversary must follow to successfully mount the attack so we can build preventive security measures. Using BFI as a core, we created *RoboTack*, an intelligent malware that helps assess a system's security and reliability (Chapter 4). Contrary to current techniques to counter adversarial attacks, our approach focuses on evaluating the end-to-end dependability of AVs instead of focusing *only* on ML/AI models (e.g., stop-sign attacks). A key feature of RoboTack is its ability to disguise attacks as accidental/random to evade detection yet cause serious safety/reliability incidents (e.g., an accident of an autonomous vehicle). RoboTack does this by answering the questions of *what*, *how*, and *when* to attack the system being tested by using a runtime decision framework whose goal is to decrease the safety potential within some threshold duration. The neural network uses the telemetry data to identify the most vulnerable system state (answering *when*) and the corresponding faults (answering *what*) that will minimally perturb the system (i.e., without being detected) while still leading to safety/reliability incidents (answering *how*). RoboTack demonstrates the steps of an attack that can be used by an adversary to leverage known faults and failure modes.

A RoboTack-generated attack is highly fatal (15–25× more likely to be fatal than state-of-the-art adversarial attacks [26]), and it evades known detection techniques.

Reducing runtime overhead of detecting hardware failures. Computational elements, such as CPUs, GPUs, and ASICs, used in autonomous vehicles (AVs) are susceptible to transient or permanent hardware faults. Faults may lead to a detectable, uncorrectable error (DUE) that degrades system availability. Practical implementation autonomous driving systems include a fail-back system that maintains the safety of the system in the case of a DUE. In contrast, an undetected error, such as a silent data corruption (SDC), may cause faulty vehicle behavior that may lead to significant safety hazards, resulting in loss of human life and serious damage to vehicles [27, 6, 24]. Future trends of increasing code complexity and shrinking feature sizes will only contribute to increasing the failure rate, thereby exacerbating the problem. Thus, detecting and mitigating SDCs caused by hardware faults is important.

We developed, DiverseAV, a novel alternative to full duplication to detect transient and permanent hardware faults that offers high error detection coverage with low performance overheads (<25%), along with corresponding power savings (Chapter 5). Our approach requires no additional hardware and minimal modification of the AV software. DiverseAV is a lightweight, software-based redundancy technique that exploits the temporal data diversity present in the sensor data for detecting hardware faults. DiverseAV creates two redundant data-diverse agents by distributing the sensor data between the

two agents in round-robin. The sensor data obtained between the two consecutive sequential time steps is semantically similar in terms of worldview but significantly different at the bit-level, ensuring state and data diversity between the two agents. The data-diverse agents use the same underlying agent models (and software code) and are together responsible for driving the AV. The outputs produced by the two agents are close to each other in the fault-free case. However, in presence of a safety-critical fault, the outputs diverge significantly; thereby, enabling safety-critical fault detection. Since much of the data processing in each agent depends on the input data rate, each agent receives half the data and requires roughly half the compute resources. Thus, in DiverseAV, we time-multiplex the two agents on the shared computational fabric.

DiverseAV detected safety-critical errors caused by the transient and permanent faults injected into the computational elements with a precision of 0.87 and a recall (which is equivalent of the detection coverage) of 0.87. DiverseAV outperformed both a fully-duplicated system and a single agent system (which uses temporal outlier detection techniques) in terms of accuracy.

Assessing and managing risk at runtime. Driving in a dynamic environment with other actors is inherently a risky task, as each actor influences driving decisions and may significantly limit the number of choices in terms of navigation and safety plan. The risk encountered by the Ego actor (i.e., AI-agent under the test) depends on the driving scenario and the uncertainty associated with predicting the future trajectories of the other actors (NPCs) in the driving scenario. However, not all NPCs pose a similar risk. Depending on the NPC's type, trajectory, position, and the uncertainty associated with these quantities, some NPCs pose a much higher risk than others. The higher the risk associated with an NPC, the more attention must be directed towards that NPC in terms of resources and safety planning.

In Chapter 6, we propose a metric that captures the *importance* of each NPC in the world with respect to their potential to create a safety hazard. In particular, the *importance* metric characterizes the decrease in Ego actor's driving flexibility with respect to a given NPC or driving scenario. The more constrained the Ego actor, the higher the chance of a safety hazard, and therefore, the higher is the risk. By characterizing a real-world dataset using our metric, we find that $<$0.1% of NPCs in the environment constrain the Ego actor. We propose a novel neural-network-based model to estimate the *importance* metric at runtime with significantly less overhead in terms of compute and memory while meeting the deadline requirements for runtime monitoring. Moreover, we show that integrating the *importance* metric with offline assessment techniques eliminates the need to test the adverse effect of faults or attacks for all NPCs, as only few NPCs are important

at any given time.In this way we reduce the test set and provide up to 24× acceleration over the current state-of-the-art assessment techniques.

1.4 AUTOMATED HIGH-PERFORMANCE COMPUTING SYSTEMS

Driven by the needs of exponentially increasing computation demands for emerging scientific and commercial applications [28], large-scale computing systems such as HPC and cloud computing systems are increasing becoming large, complex, and heterogeneous by incorporating innovations in hardware architecture, operating systems, network interconnects, and storage. This increase in complexity, heterogeneity, and scale necessitates the use of automation techniques across the system stack using heuristic-based and ML/AI methods, as is evident from recent work [29–32, 30, 33–35]. Computer systems experience a wide range of *failure modes* [1], such as fail-stop [36, 37], partial-failure [38], fail-slow [39, 40], and intermittent failure [41], including resource overload and congestion [42–45]. Such failures lead to service-level incidents [46, 47], local and system-wide outages [36], or application failures [37] and slowdown [44, 48] However, current automation techniques are tuned to handle average-case performance and fail-stop failures on a per-component/subsystem basis rather than end-to-end system performance and reliability; thus, leading to scaling issues and loss of useful computational hours. Moreover, these automation techniques work independently of one another and misdiagnose the root-cause, leading to incorrect mitigation techniques that often worsen the overall performance and reliability of systems and applications.

Characterizing production datasets. Empirical field-failure and performance characterizations to date have typically been gross breakdowns of occurrence by identified component type (memory, voltage regulator, CPU, etc.) and/or by failure type (hardware, software, network, human, etc.), system wide over a time period. Frequently, the studies result in presenting failure rate statistics. This type of characterization, while useful, does not provide sufficient fidelity or understanding to enable continuous assessment and mitigation of dependability problems. This problem exists because of several challenges, which include: (i) dealing with the volume, velocity, and veracity of data and (ii) developing models that enable holistic understanding of the impact of local failures on end-to-end system dependability.

To address these challenges, we develop domain-driven models using data to capture the relationship between the end-to-end system dependability and various failure modes. We collected monitoring datasets from Blue Waters [49–51], which is situated at the Na-

tional Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. Blue Waters is one of the largest supercomputers at an academic institution in terms of node count, network, and storage size. This thesis specially focused on network- and storage-related failures and their impact on application and system performance, which are outlined in Chapter 7 and Chapter 8, respectively.

These studies highlight the issue of misdiagnosis of failure modes. Especially, we find that issues related to reliability failures and performance anomalies are hard to disambiguate, leading to significant performance loss and scaling problems. For example, I/O requests during reliability failures increase the average completion time of I/O requests by up to 52.7×compared to the average I/O completion time in failure- free scenarios (approximately 200ms). Such issues exist because failure monitors and automated mitigation techniques operate independently from one another and often cannot address problems that cut across the system stack.

Detecting, isolating and diagnosing failures at runtime. Using the insights from failure data, we developed Kaleidoscope [48] to detect various failure modes at runtime proactively (Chapter 8). In addition to targeting good failure detector properties, such as *completeness* and *accuracy*, we target (i) *localization*, which pinpoints the location of the failure at the lowest possible failure containment boundary and (ii) *differentiability*, which enables identification of the failure mode.

Kaleidoscope leverages existing hierarchical monitors and uses ensembles of domain-guided ML models for detection and disambiguation of failure modes. Kaleidoscope uses probabilistic graphical model (PGM) formalism to jointly analyze and fuse the telemetry dataset from across the monitors. In this model, the state of the entire system is represented by a joint distribution over the health state of each component. An inference on this model using the observations from the monitoring data is used to estimate the health of each component while accounting for the noise and related uncertainties in the data. This determination of the failure state localizes failed components in the system. Once the failed component(s) is determined, Kaleideoscope uses an unsupervised machine-learning method, the local outlier factor [52], to identify the anomalous feature (and hence, the failure mode) that best distinguishes the failed component(s) from the healthy ones.

Kaleidoscope is a scalable approach (tested on Blue Waters with 25K+ nodes) that is highly accurate in detecting (99.3% accurate) and disambiguating (95.3% accurate) failure modes, with low overhead (impact is $<$0.01% of peak I/O bandwidth). Moreover, Kaleidoscope can be used as a long-term characterization tool to inform design decisions.

1.5 PRACTICAL DEPLOYMENT & INDUSTRY ADOPTION

Adoption of assessment tools. Our work on assessing autonomous vehicles, using Bayesian Fault Injector (BFI) [24] to identify vulnerabilities, grabbed worldwide attention, especially in the USA and China as evident from worldwide press coverage: Science Daily [53], Daily Illini [54], Guancha.cn [55], and Space Daily, among others. BFI was used to evaluate two production-grade, self-driving autonomous driving systems: (i) NVIDIA's DriveAV and (ii) Baidu's Apollo. It also gained attention from other industry members such as LG, Qualcomm, Samsung, and Intel.

Similar to BFI, we developed HPCArrow [56] for assessing high-performance networking interconnects. HPCArrow has been used on two large-scale, high-performance computers (commonly referred to as supercomputers): Edsion (> 5000 nodes) at National Energy Research Scientific Computing Center (NERSC) and Cielo (> 10000 nodes) at Los Alamos National Lab (LANL). To date, our fault injection campaign remains one of the largest efforts to assess the network interconnect of production systems. HPCArrow was also used to evaluate other high-performance computing systems at Sandia National Laboratories and National Center for Supercomputing Applications.

Adoption of online monitoring and diagnostic tools. We developed Kaleidoscope [48], an ML-driven monitoring and diagnostic tool, which uses probabilistic graph models and causal principles. Ideas from Kaleidoscope are being used in production at NCSA Blue Waters for monitoring and diagnosing file system failures. We also tested our ideas on IBM Cloud for monitoring and diagnosing customer incidents.

The proposed techniques also serve as a basis on which we developed techniques for mitigating service-level objective (SLO) violations for microservices [57], networks [42], and storage [48]. Additionally, the developed technique was combined with reinforcement learning for designing an ML-driven scheduler. Across these domains, we improved the system dependability by up to $100\times$ and reduced performance anomalies by $2 - 20\times$.

1.6 BROADER IMPACTS

Society as a whole is going to witness exponential growth in the adoption of AI/ML-driven systems in critical application domains such as healthcare, transportation, agriculture, and manufacturing. The availability and deployment of dependable AI-driven systems are valuable because they (i) increase efficiency of tasks carried out by humans (e.g., search and rescue missions, package delivery, and healthcare) and (ii) enable execution

of tasks that are nearly impossible or dangerous for humans (e.g., mineral mining and deep-sea exploration). The widespread adoption of such systems in a human-centric environment necessitates the understanding of AI-engineered systems and their capabilities in the presence of a wide range of uncertainties from specification to real-time operations. These next-generation AI-driven systems demand an ever-increasing level of system dependability (i.e., performance, robustness, security, maintainability, and ease of use) not available today. The classical approach to dependability (availability, fault tolerance, integrity, security, etc.) is based upon component reliability views and fault/error/attack management at the architecture level. While necessary, the classical approaches are not sufficient, and new methods must be developed to account for autonomy and safety requirements.

This thesis is a step in that direction. We develop novel causality-driven techniques that meet those demands and provide the theory and foundation for designing dependable automated and autonomous systems. Methods proposed in this thesis will allow designers to assess and ensure the dependability of such systems. We showcased these techniques on complex mission-critical automated and autonomous systems: (i) autonomous vehicles (particularly, self-driving cars) and (ii) management of large-scale computing infrastructures (HPC and Cloud). Techniques proposed in this work will pave the path to ensuring the dependability of other autonomous systems, such as unmanned aerial vehicles, agricultural robots, and kitchen bots, among others.

CHAPTER 2: AV: FIELD MEASUREMENTS

Autonomous vehicle (AV) technology is rapidly becoming a reality on U.S. roads, offering the promise of improvements in traffic management, safety, and the comfort and efficiency of vehicular travel. The California Department of Motor Vehicles (DMV) reports that between 2014 and 2017, manufacturers tested 144 AVs, driving a cumulative 1,116,605 autonomous miles, and reported 5,328 disengagements and 42 accidents involving AVs on public roads. This chapter investigates the causes, dynamics, and impacts of such AV failures by analyzing disengagement and accident reports obtained from public DMV databases. We draw several conclusions. For example, we find that autonomous vehicles are $15 - 4000\times$ worse than human drivers for accidents per cumulative mile driven; that drivers of AVs need to be as alert as drivers of non-AVs; and that the AVs' machine-learning-based systems for perception and decision-and-control are the primary cause of 64% of all disengagements.

2.1 INTRODUCTION

Autonomous vehicle (AV) technologies are advertised to be transformative, with a potential to improve traffic congestion, safety, productivity, and comfort [58]. Several states in the U.S. (e.g., California, Texas, Nevada, Pennsylvania, and Florida) have already started testing AVs on public roads. Prior research into AVs has focused predominantly on the design of automation technology [59–64], its adoption [65], the impact of AVs on congestion[66], and the legal [67, 68] and regulatory barriers [69–72] for AV implementation. With the increasing popularity and ubiquitous deployment of semi- and fully-automated vehicles on public roads, safety and reliability have increasingly become critical requirements for public acceptance and adoption. This chapter assesses, in broad terms, the reliability of AVs by evaluating the cause, dynamics, and impact of failures across a wide range of AV manufacturers utilizing publicly available field data from tests on California public roads, including urban streets, freeways, and highways.

Dataset. The California Department of Motor Vehicles (CA DMV) mandates that all manufacturers testing AVs on public roads file annual reports detailing *disengagements* (a failure that causes the control of the vehicle to switch from the software to the human driver) and *accidents* (an actual collision with other vehicles, pedestrians, or property) [23]. The focus of the testing program, and of this chapter, is on semi-autonomous vehicles that require a human driver to serve as a fall-back in the case of failure. In partic-

ular, we are interested in studying failures that pertain to sensing (e.g., cameras, LIDAR) and computing systems (e.g., hardware and software systems that enable environment perception and vehicle control) that enable the "self-driving" features of the vehicles. We analyze field data collected over a 26-month period from September 2014 to November 2016 (part of the DMV's 2016 and 2017 data releases), containing data from 12 AV manufacturers for 144 vehicles that drove a cumulative $1,116,605$ autonomous miles. Across all manufacturers, we observe a total of $5,328$ disengagements, 42 of which led to accidents.

Results. This chapter presents 1. an end-to-end workflow for analyzing AV failure data, and 2. several insights about failure modes in AVs (across a single manufacturer's fleet, across different manufacturers, and in time) by executing the proposed workflow on the available data. Our study shows:

1. Drivers of AVs need to be as alert as drivers of non-AV vehicles. Further, the small size of the overall action window (detection time + reaction time) would make reaction-time-based accidents a frequent failure mode with the widespread deployment of AVs.
2. For the same number of miles driven, for the manufacturers that reported accidents, human-driven non-AVs were $15 - 4000\times$ less likely than AV's to have an accident.
3. 64% of disengagements were the result of problems in, or untimely decisions made by, the machine learning system.
4. In terms of reliability per mission, AVs are $4.22\times$ worse than airplanes, and $2.5\times$ better than surgical robots.

These findings demonstrate that while individual components of AV technology (e.g., vision systems, control systems) may have matured, entire AV systems are still in a "burn-in" phase.

The analysis presented in this chapter shows a distinct improvement in the performance of AVs over time. However, it also demonstrates the need for continued improvement in the dependability of this technology. It is conceivable (moreover, expected) that AV manufacturers are performing a similar analysis of data coming from their testing fleets, but to the best of our knowledge, information on such analysis is not available publicly. Our goal is to support resilience research by characterizing failures of autonomous vehicles, rather than to further the operational perspective of the manufacturer. Our results can better inform the design of future AVs.

Organization. Fig. 2.1 shows the end-to-end pipeline for processing failure data from autonomous vehicles. §2.2 describes two real examples of AV-related accidents on California roads. §2.3 describes the AVs and the data collection methodology (*Stage I* of the pipeline). §2.4 describes the preprocessing, filtering, and natural language processing (NLP) steps required to convert the data to a format suitable for analysis (*Stages II & III* of

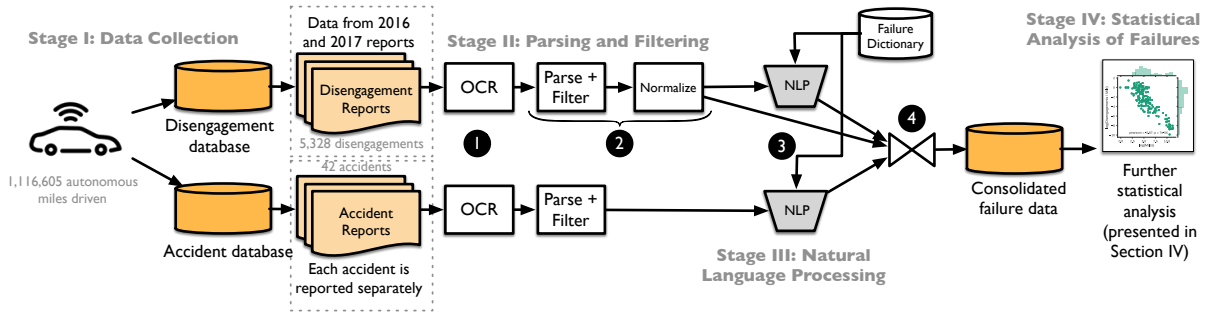


Figure 2.1: The end-to-end data collection, processing, and analysis pipeline that forms the basis of this study.

the pipeline). §2.5 describes the statistical analysis of the failure data and summarizes the insights derived from the analysis (*Stage IV* of the pipeline). Finally, §2.6–§2.8 describe the threats to validity, related work and conclusions, respectively.

2.2 CASE STUDIES

In this section, we present two representative case studies based on real events that occurred in the streets of Mountain View, CA. These case studies illustrate how problems in the perception, learning, and control systems of an AV can manifest as an accident.

2.2.1 Case Study I: Real-Time Decisions

Example ❶ in Fig. 2.2 shows a case in which the human driver of the AV proactively took over the control of the vehicle from the autonomous agent (to prevent an accident) but was unable to rectify decisions made by the autonomous agent in time to prevent an accident. The disengagement report (i.e., error logs from the AV combined with post-mortem analysis performed by the manufacturer) logs the error as either "`Disengage for a recklessly behaving road user`" or "`wrong behavior prediction`." Specifically, a Waymo prototype vehicle was in autonomous mode at a street intersection when a pedestrian started to cross the street. From the accident report, we find that the AV decided to yield to the pedestrian but did not stop. The test driver proactively took control of the car as a precaution. At the same time, there was a car in front of the AV that was also yielding to the pedestrian, and another vehicle to the rear in the adjacent lane that was making a lane change. In this complex scenario, the driver did not have many options other than to brake, and the rear vehicle collided with the back of the AV.

2.2.2 Case Study II: Anticipating AV Behavior

Example 2 in Fig. 2.2 shows a case in which a Waymo prototype vehicle was running in autonomous mode and was hit by a manual vehicle from the rear at a street intersection. The disengagement report logs the cause as "`Disengage for a recklessly behaving road user`." In this case, the AV had signaled a right turn and had started to decelerate for the turn. It came to a complete stop before it started moving again towards the intersection to gauge the traffic coming from the other side in order to make a safe turn. The movement towards the intersection was required to allow the recognition system to analyze the scene and produce a movement plan for the car. The driver of the rear vehicle was confused and interpreted this movement to mean that the AV was conitinuing on its path (i.e., making the turn). The driver first stopped (as the AV stopped) and then started moving (as the AV started to move again). This resulted in a rear collision on the AV, as the driver could not anticipate the actions of the AV.

2.2.3 Summary

By law, both of those accidents were caused by the drivers in the non-AV; however, close inspection of the accident reports shows that the AV had a significant share of the responsibility. The above examples showcase the poor AV decision-making that eventually leads to accidents.

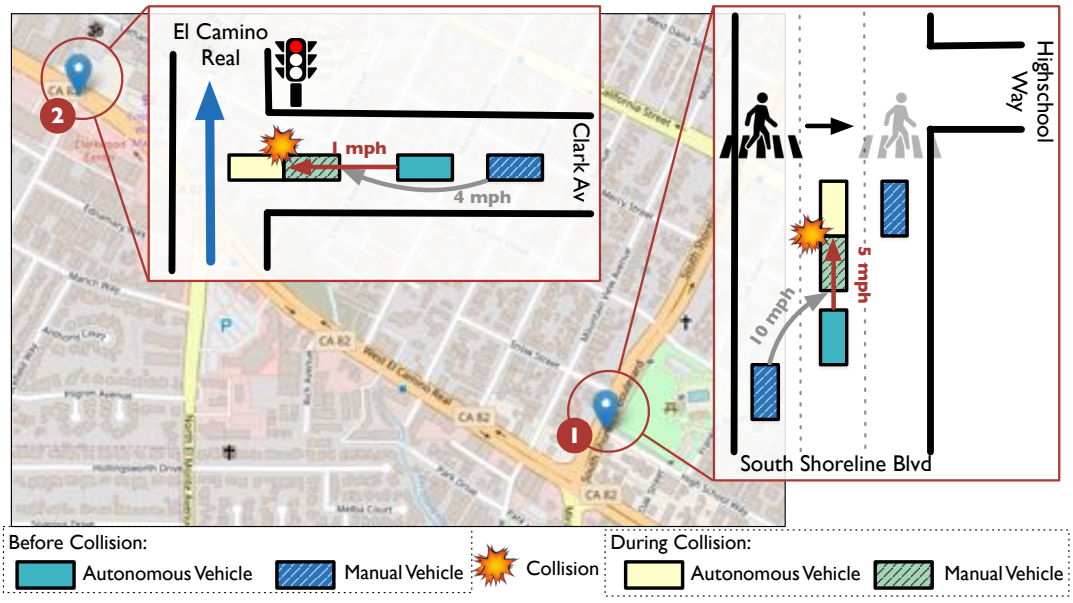


Figure 2.2: Accident scenarios.

1. The street intersections represent complex scenarios in which the AV needs to analyze multiple traffic flows and make decisions in a constrained environment. Based on our analysis we attribute the failures to the learning-based perception system, which did not infer in time the evolving environment dynamics from the onboard sensor systems (e.g., RADAR, LIDAR), leading the learning-based control system to make inadequate decisions.
2. In both cases, drivers either voluntarily took or were forced to take control from the autonomous system in complex and dynamic traffic scenarios that frequently gives them very little time to react and undo the AV's actions. The perception and reaction time is crucial in accident avoidance.
3. Drivers in other non-AVs often cannot anticipate decisions made by AVs, which frequently also leads to accidents.

Using the limited publicly available information about the design of the AV systems (e.g., [73–76]), we draw our conclusions by analyzing human-entered textual logs that contain information about accidents and disengagements. Our method localizes failures to the learning, perception, and decision-and-control subsystems of an AV to understand the causes of disengagements and accidents.

2.3 AV SYSTEM DESCRIPTION AND DATA COLLECTION

2.3.1 Preliminaries

Autonomous Vehicles

An AV is any vehicle that uses an autonomous driving system (ADS) technology capable of supporting and assisting a human driver in the tasks of 1. controlling[1] the main functions of steering and acceleration, and 2. monitoring the surrounding environment (e.g., other vehicles/pedestrians, traffic signals, and road markings) [77].

The Society of Automotive Engineers (SAE) defines six levels of autonomy that are based on the extent to which the technology is capable of supporting and assisting the driving tasks [77]. The levels of autonomy go from 0 (no automation) to 5 (full, unrestricted automation). Levels 0–2 (e.g., anti-lock braking, cruise control) require a human driver to be responsible for monitoring the environment of the vehicle, with different levels of automation available to support vehicle control tasks. Levels 3–5 are thought of

[1]Here, "control" incorporates both decision and control.

as truly automated driving systems where the AV both monitors the environment and controls the vehicle. The subject of this chapter is the Level 3 vehicles.

Disengagements

Level 3 requires the presence (and attention) of a human driver to serve as a fall-back when the autonomous system fails. A transfer of control from the autonomous system to the human driver in the case of a failure is called a *disengagement*. Disengagements can be initiated either manually by the driver or autonomously by the car. Manual disengagements initiated by the driver are cautionary (e.g., if one feels uncomfortable, or wants to adopt a proactive approach to prevent a potential accident). Automated disengagements are indicative of a design limitation of the AV.

Accidents

An *accident* is an actual collision with other vehicles, pedestrians, or property. Note that not all disengagements lead to collisions. As we show later in this chapter, most disengagements are handled safely by the human operators, with only a small fraction leading to accidents. For example, in some reported collisions, the test driver initiated a manual disengagement before the collision (an artifact of the training program that all test drivers acting as AV safety-pilots have to undergo before they are allowed on public roads [23]).

2.3.2 AV Hierarchical Control Structure

Manufacturers have not disclosed the architectures of their autonomous vehicles. However, to identify multidimensional causes of AV disengagements/accidents, we built a hierarchical control structure for AVs by using the systems-theoretic hazard modeling and analysis abstraction STPA (Systems-Theoretic Process Analysis) [78]. Fig. 2.3 shows an AV hierarchical control structure derived based on technical documentation [18–22]. We assert that these information sources are representative and provide a conceptual view of AV systems that is sufficiently detailed to enable creation of an STPA model. We refer to this system as the "Autonomous Driving System" (ADS). The major components of the ADS are 1. "sensors" (e.g., GPS, RADAR, LIDAR, and cameras) that are responsible for collecting environment-related data, 2. a "recognition system"[2] that uses sensor data

[2]The "recognition system" is also referred to as the "perception system."

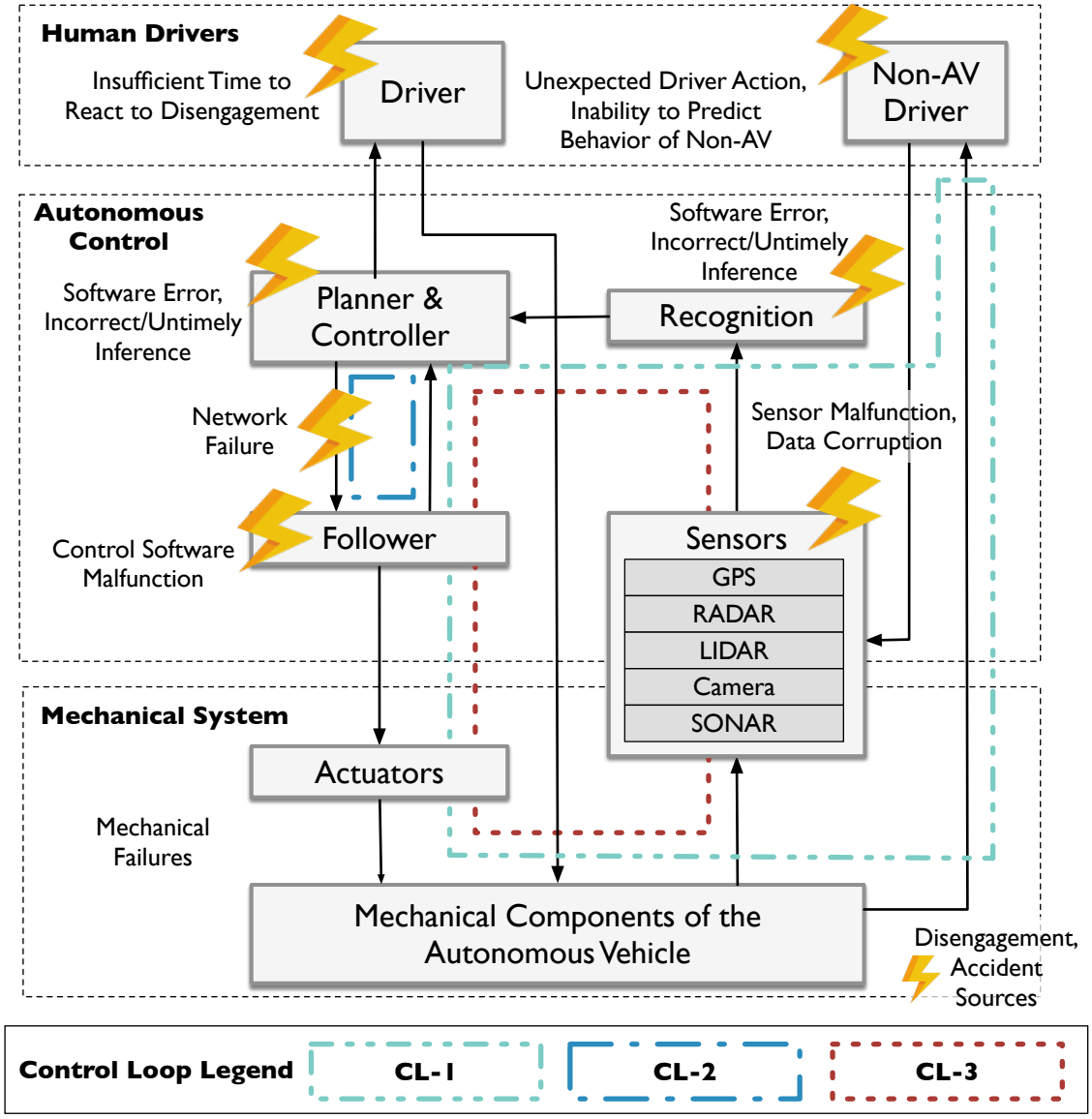


Figure 2.3: Autonomous vehicle hierarchical control structure drawn based on [18]. Examples of control loops are highlighted as `CL-1`, `CL-2`, and `CL-3`.

to identify the objects and changes in the environment around the AV, 3. a "planner and controller" system that is responsible for planning the next motion of the car based on the current parameters of the AV and the environment (e.g., speed, location, and other vehicles), and 4. a "follower" system that signals the "actuators" to drive the vehicle along the path chosen by the "planner and controller."

STPA employs concepts from systems and control theories to model hierarchical control structures in which the components at each level of the hierarchy impose safety constraints on the activity of the levels below and communicate their conditions and behavior

to the levels above them. Accidents and disengagements are complex dynamic processes resulting from inadequate perception control and decision-making at different layers of the system control structure. Accidents and disengagements seen in the data were overlaid on this structure.

In every control loop, the planner and controller system uses an algorithm to generate the control actions based on a model of the current state of the process that it is controlling. The control actions (e.g., "decelerate") taken by the planner and controller system (i.e., the autonomous driving system) change the state of the controlled process (e.g., mechanical components of the autonomous vehicle). The feedback message (e.g., the state of the traffic lights) sent back from the controlled process (e.g., the AV control software) updates the process model used (e.g., the mental model the driver has of the AV status) by the controller. Analysis of dependencies along those control loops allows for the identification of inadequate controls and the potential causes of those unsafe control actions through examination of the operation of components and their interactions in each loop of the control structure. Any flaws or inadequacies in the algorithm, the process model, or the feedback used by a controller are considered potential causal factors leading to unsafe control actions and resultant disengagements/accidents.

In Fig. 2.3 we highlight three control loops (`CL-1`, `CL-2`, and `CL-3`, indicated with different types of dashed lines) to illustrate details of the interactions among the driver (both AV and Non-AV), AV control, and AV hardware/software components. Our analysis couples that STPA approach with manufacturers' reports. The most complex control loop, `CL-1`, involves interaction among the *autonomous control* (including sensors, recognition system, planner, and controller), *mechanical system* (actuators and mechanical components of the vehicle), and *human drivers* (drivers of non-AVs). The Non-AV Driver module represents the AV system's ability to 1. collect the data on Non-AV driver behavior through the sensors, and 2. provide information (e.g., on brake signals, turn indicators, or horn) to Non-AV drivers. Examples of failures in this control loop were discussed in the two case studies presented earlier.

Table 2.1: Summarization of fleet size, autonomous miles driven, and failure incidents across all manufacturers in the dataset.

	2015–2016 Report				2016–2017 Report			
Manufacturer	**Cars**	**Miles**	**Disengagments**	**Accidents**	**Cars**	**Miles**	**Disengagements**	**Accidents**
Mercedes-Benz	2	1739.08	1024	–	–	673.41	336	–
Bosch	2	935.1	625	–	3	983	1442	–
Delphi	2	16661	405	1	2	3090	167	–
GM Cruise	–	285.4	135	–	–	9729.8	149	14
Nissan	4	1485.4	106	–	3	4099	29	1
Tesla	–	–	–	–	5	550	182	–
Volkswagen	2	14946.11	260	–	–	–	–	–
Waymo (Google)	49	424332	341	9	70	635868	123	16
Uber ATC	–	–	–	–	–	–	–	1
Honda	–	–	–	–	0	0	0	–
Ford	–	–	–	–	2	590	3	–
BMW	–	–	–	–	–	638	1	–
Total	61	460384.1	2896	10	83	656221	2432	32

Dashes indicate the absence of data in the manufacturer's report.

2.3.3 Data Sources

The CA DMV is the state agency that registers motor vehicles, issues regulations and permits, and monitors the testing and field operation of autonomous vehicles. California driving conditions are representative of urban situations and the DMV has a strong mandate for data collection and public availability. California law requires the manufacturers operating and testing AVs to file reports on disengagements (reported annually) and accidents (reported within ten business days of the incident) [23, 79]; these reports are eventually made public. The reports are available as a part of two databases:

1. *AV Disengagement Reports*: These reports contain aggregated information about fleet size, monthly autonomous miles traveled, and the number of disengagements observed. Each manufacturer provides its own data format, resulting in a fragmented set of data. Some manufacturers provide additional information, including timestamps, road type (e.g., urban streets, highway, freeway), weather conditions (e.g., sunny, raining, overcast), driver reaction times (time taken for the driver to disengage from autonomous mode), and other factors contributing to the disengagements. We use the additional data whenever it is available.
2. *AV Accident Reports*: These reports contain timestamped information about the autonomous vehicle involved, the location of the accident, descriptions of other vehicles involved (e.g., class of vehicle, speed), and human-written textual description of the incident and its severity.

Both datasets consist of scanned documents containing both tabulated data and natural-language text. Unlike previous analyses [80, 81], which are based solely on the data provided, we focus on building an analysis workflow that processes substantive amounts of human-generated disengagement and accident reports by using NLP.

Summary of Datasets. The datasets cover 12 AV manufacturers (Bosch, Delphi Automotive, Google, Nissan, Mercedes-Benz, Tesla Motors, BMW, GM, Ford, Honda, Uber, and Volkswagen). With 144 AVs that drove a cumulative $1,116,605$ autonomous miles across 9 distinct road types (31.7% on city streets, 29.26% on highways, 14.63% on interstates, 9.75% on freeways, and the remaining 14.6% in parking lots, suburban, and rural roads). Uber, BMW, Ford, and Honda reported too few disengagements for us to draw statistically significant conclusions, so are left out of the analysis in this chapter. Across all manufacturers, we observe a total of $5,328$ disengagements[3] and 42 accidents (including the two case studies in §2.2). Aggregating per car and per manufacturer, we observe

[3]Two of the manufacturers (Bosch and GMCruise) reported all their disengagement data as planned tests. Our understanding, based on all the DMV reports, is that the tests were planned, but the disengagements occurred naturally. Together the two manufacturers have 14 accidents during "tests".

Table 2.2: Sample of disengagement reports from the CA DMV dataset.

Manufacturer	Raw Disengagement Report (Log)	Category	Tags
Nissan	1/4/16 — 1:25 PM — **Software module froze**. As a result driver safely disengaged and resumed manual control. — City and highway — Sunny/Dry	System	Software
Nissan	5/25/16 — 11:20 AM — Leaf #1 (Alfa) — The AV **didn't see** the lead vehicle, driver safely disengaged and resumed manual control.	ML/Design	Recognition System
Waymo	May-16 — Highway — Safe Operation — Disengage for a **recklessly behaving** road user	ML/Design	Environment
Volkswagen	11/12/14 — 18:24:03 — Takeover-Request — **watchdog error**	System	Computer System

We use the "—" to denote field separators.
Note that log formats vary across manufacturers and time.
Bold-face text represents phrases analyzed by the NLP engine to categorize log lines.

an average of 262 autonomous miles driven per disengagement, and one accident event for every 127 disengagements.

Across manufacturers in the dataset, we observe a significant skew in the number of autonomous miles driven (see Table 2.1). For example, Waymo tested their AV prototypes more extensively than the others (over 1,000,000 miles compared to 15,000 miles for the next highest testing manufacturer). This suggests that Waymo's AVs might perform better than those of its competitors because of the extensive testing of the ADS platform. Note that not all manufacturers provide all the data needed to compute the summary statistics; those omissions are indicated by dashes in Table 2.1.

2.4 DATA-ANALYSIS WORKFLOW: PARSING, FILTERING, NORMALIZATION AND NLP

Fig. 2.1 describes our methodology (workflow) for converting raw disengagement and accident reports into a consolidated form that lends itself to further analysis. Below, we describe the key steps involved in Stages II and III of the workflow.

Table 2.3: Definition of fault tags and categories that are assigned to disengagements.

Tag	Category	Definition
Environment	ML/Design	Sudden change in external factors (e.g., construction zones, emergency vehicles, accidents)
Computer System	System	Computer-system-related problem (e.g., processor overload)
Recognition System	ML/Design	Failure to recognize outside environment correctly
Planner	ML/Design	Planner failed to anticipate the other driver's behavior
Sensor	System	Sensor failed to localize in time
Network	System	Data rate too high to be handled by the network
Design Bug	ML/Design	AV was not designed to handle an unforeseen situation
Software	System	Software-related problems such as hang or crash
AV Controller	System	"System" when AV controller does not respond to commands
	ML/Design	"ML/Design" when AV controller makes wrong decisions/predictions
Hang/Crash	System	Watchdog timer error

Digitization of the Accident and Disengagement Reports. The aforementioned logs are provided in the form of scanned images of digital documents (for disengagement reports) and hand written reports (for accident reports). The first task is to pre process and convert these scanned reports into a machine-encoded format. Examples of such machine-encoded disengagement reports are shown in Table 2.2. Hence, our analysis proceeds with optical character recognition (OCR; labeled as ❶ in Fig. 2.1) by using Google Tesseract [82] on the scanned documents. In certain cases, where the Teserract OCR failed (because of low-resolution scans or inability to recognize some table formats), we manually converted the documents to machine-encoded text.

Data Normalization. CA DMV regulations require that each manufacturer report crucial information about disengagements, e.g., the number of miles driven in autonomous mode and the number of disengagements observed. However, it does not enforce any data format specification for these reports, leading to disparities (across manufacturers

and across time) in the data schema and granularity of the information available through these reports. Hence, we need to filter, parse, and normalize (labeled as ❷ in Fig. 2.1) the data into machine-encoded text to produce structured datasets that have uniform schema across manufacturers and time (i.e., across reports made by the same manufacturer at different times). Taken together, steps ❶ and ❷ correspond to preprocessing of the datasets to make them ready for further analysis.

Labeling and Tagging of the Reported Disengagement and Accident Causes. The pipeline uses an NLP-based technique (labeled as ❸ in Fig. 2.1) to map a given disengagement event in a corresponding fault tag and a failure category. First we make several passes over the dataset to construct a "Failure Dictionary" that contains a sequence of phrases (keywords) extracted from the raw disengagement reports (logs). This dictionary is used to design a voting scheme (which is based on the maximum number of shared keywords) to assign a disengagement cause to a fault tag. In the event that this procedure is unsuccessful and we cannot associate any of the known tags to textual description, the disengagement cause is marked with the "Unknown-T" tag.

We then build an ontology (based on Fig. 2.3) of failure *categories* on top of the tags (which were derived from [83]). Specifically, we apply our understanding of the ADS system (described in §2.3.2) to select keywords and phrases that differentiate fault tags from each other. The tags are chosen to localize faults in the computing system (e.g., software and hardware systems) and in the machine learning algorithms/design (e.g., perception and control algorithms), thereby identifying potential targets for improving the safety and reliability of the AV. Table 2.3 lists the fault tags used in this study. Table 2.2 provides examples of the raw log to tag and category mapping. We consider the following failure categories: 1. faults in the design of the machine learning system responsible for "perception" tasks (dealing with data from sensors) and "planning and control" tasks (dealing with control of steering and acceleration); 2. faults in the computing system (dealing with hardware and software problems); and 3. an "Unknown-C" category consisting of tags we cannot classify into any of the above categories.

These tags and categories allow us to classify the types of failure causes into machine-learning vs. computer-system-related issues. Table 2.3 provides a mapping between the categories and tags used in our analysis. In the final step (labeled as ❹ in Fig. 2.1), the preprocessed data from the disengagement dataset and accident dataset are merged together, along with extracted categories and tags, to create a *consolidated* AV failure database.

2.5 STATISTICAL ANALYSIS OF FAILURES IN AVS

Traditional approaches to evaluating the resilience of a system [1] require computation

of *availability*, *reliability*, and *safety*. These metrics require information about operational periods of the AV (e.g., the active time of the vehicle). As this information is not available in the CA DMV dataset, we use the 5,324 disengagements (across eight manufacturers) and 42 accidents as the basis for deriving statistics on fault classes, failure modes of AVs, and their evolution over time. These statistics allow us to draw conclusions and answer the following questions:

Question 1. How do we assess the stability/maturity of the AV technology?

Question 2. What is the primary cause of disengagements (and potentially accidents) observed in AVs?

Question 3. Are manufacturers indeed building better and more reliable AVs over time?

Question 4. What level of alertness[4] of the human driver of an AV guarantees safety?

Question 5. How well do AVs compare with human drivers?

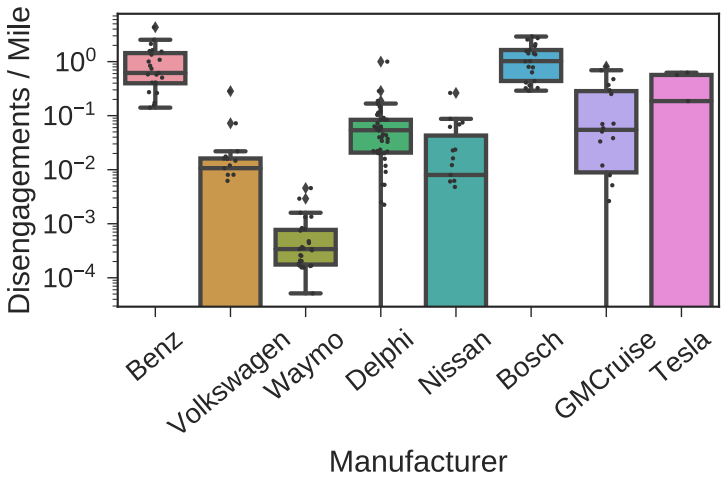


Figure 2.4: Comparison of the distributions of DPM per car across manufacturers. The boxes show quartiles; the notches show medians; and the whiskers show max/mins.

2.5.1 Analysis of AV Disengagement Reports

Question 1: Assessment of AV Technology

Based on the available data, we computed the following metrics from the disengagement reports to assess AVs: 1. number of disengagements observed per autonomous mile

[4]Measured here as reaction times of human drivers in case of disengagements.

driven (DPM, shown in Fig. 2.4), and 2. total number of disengagements observed (shown in Fig. 2.5).

Comparing DPMs across Manufacturers. Most manufacturers have a median DPM $\in [0.1, 0.01]$ m^{-1} per car with the 99^{th} percentile DPM around 1 m^{-1} (see Fig. 2.4). There is a significant disparity (nearly $100\times$) between median DPMs across all manufacturers. This substantiates our initial hypothesis (from §2.3.3) that the cumulative miles driven by a manufacturer (see Table 2.1) is indicative of better performance. For example, Waymo (Google) does $\sim 100\times$ better than its competitors in terms of both the median and 99^{th} percentile DPMs; at the same time, it is responsible for $> 90\%$ of the total miles driven in the dataset.

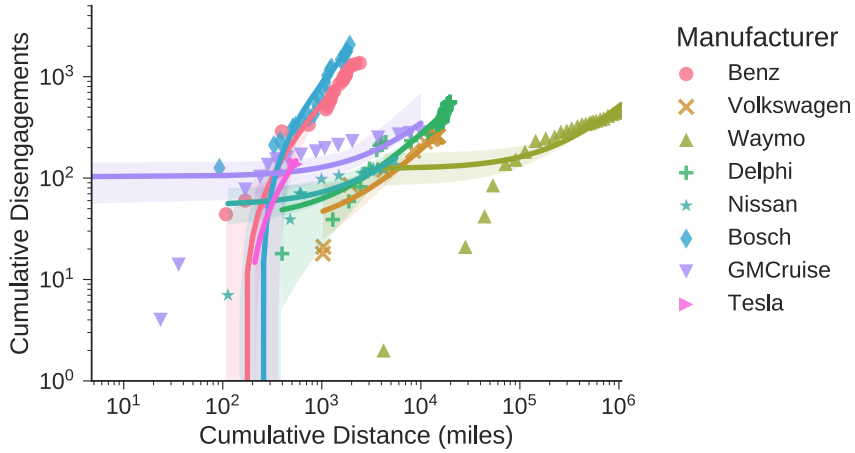


Figure 2.5: Disengagements reported per cumulative miles driven across manufacturers represented in a log-log plot. Lines represent linear regression fits.

Maturity of AV Technology. Fig. 2.5 demonstrates a strong linear correlation (based on the linear regression fits) between the number of disengagements observed and the number of cumulative autonomous miles driven. We expect that in an ideal case mature AV technology will show a decrease in DPM (i.e., the slopes of the lines in Fig. 2.5) that asymptotically reaches towards a horizontal line (or close to it, i.e., zero DPM or a very low DPM). The reason is that the data collected from the planned testing of AVs validates the computing system (e.g., by identifying software bugs) and also trains the machine learning algorithms that monitor the environment and control the steering and acceleration of the AV. Thereby eventually enabling the AVs to handle more fault scenarios, thus contributing to a decreasing DPM. This is true for most manufacturers to varying degrees with the exception of Volkswagen, Bosch, and GMCruise. *An important conclusion is that despite the million miles driven, Waymo is still not quite approaching the target asymptote. This*

Table 2.4: Disengagements across manufacturers (as percentages) categorized by root failure categories.

Manufacturer	Fault Type			
	ML/Design		System	Unknown-C
	Planner/ Controller	Perception/ Recognition		
Delphi	37.59	50.17	12.24	0
Nissan	36.3	49.63	14.07	0
Tesla	0	0	1.65	98.35
Volkswagen	0	3.08	83.08	13.85
Waymo	10.13	53.45	36.42	0

ML/Design is divided into Planner/Controller- and Perception-related problems.

indicates that Waymo and other manufacturers are still in the "burn-in" phase.

Question 2: Causes of AV Disengagements

We present a categorization of the sources of faults that cause disengagements from two different perspectives: 1. cause of occurrence, and 2. modality of occurrence.

Machine-Learning-Related Faults. First, we consider disengagements by *cause of occurrence*, i.e., categorization of the cause of a disengagement. In the following text, we ignore the numbers for Tesla, as most of their categorical label are marked "Unknown-C." We observe that machine-learning-related faults, mainly ones pertaining to the perception system (e.g., improper detection of traffic lights, lane markings, holes, and bumps), are the dominant cause of disengagements across most manufacturers. They account for $\sim 44\%$ of all reported disengagements (see Table 2.4).[5] The second major contributor to reported disengagements is the machine learning related to the control and decision framework (e.g., improper motion planning), which accounts for $\sim 20\%$ of the total disengagements. The computing system, i.e., hardware issues (e.g., problems with the sensor and processor) and software issues (e.g., hangs, crashes, bugs), accounts for $\sim 33.6\%$ of the total disengagements reported. Further, we observe that the perception-based machine learning faults are responsible for DPM measurements in the upper three quartiles. *Therefore we conclude that the faults in the perception system are directly responsible for higher DPMs across manufacturers*.

[5]We consider external fault sources such as undetected construction zones, cyclists, pedestrians, emergency vehicles, and weather phenomena (e.g., rain or sun glare) as perception-related-machine-learning related disengagements as they deal with interpretation of the environment from sensor data.

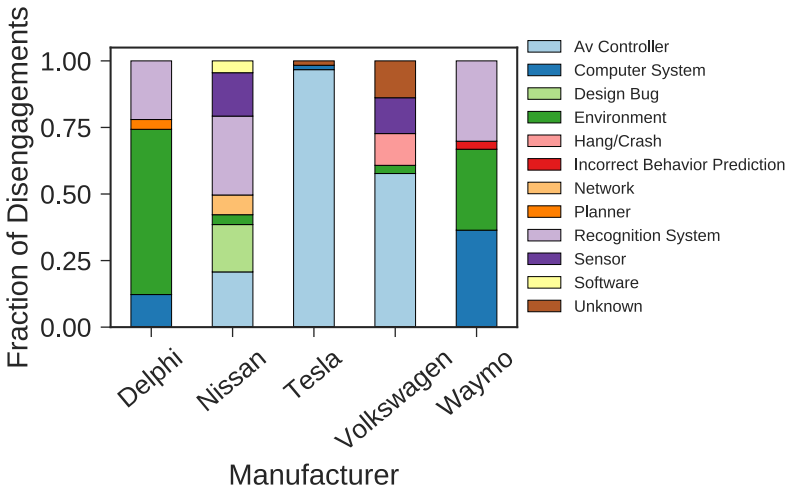


Figure 2.6: Categorization (in terms of fault tags) of faults that led to disengagements across manufacturers.

Comparing Waymo to Others Using Fault Categorization. As stated earlier, we observe that AV prototypes from Waymo perform significantly better than those of its competitors. Our fault categorization allows us to speculate on reasons for this behavior. We observe (see Fig. 2.6) that Waymo reports significantly higher percentages of disengagements related to system faults (i.e., software or hardware issues) than machine learning/design issues, unlike other manufacturers. Extensive on-road testing (over 1,060,200 cumulative autonomous miles, which is $\sim 70\times$ more than any other manufacturer) has allowed Waymo to eliminate many fault scenarios relating to perception and control. *Even though Waymo has resolved key control and decision-making issues in the machine learning system, perception and system issues still dominate. We observe that most accidents are the result of poor decisions made by the machine learning system in complex traffic scenarios, as shown in the two case studies (in §2.2). Faults in the perception systems often propagate to the decision system, leading to complex failure scenarios.*

Last, we consider disengagements by *modality of occurrence*, i.e., whether the disengagement was initiated automatically by the AV, or manually by the driver, or as part of a planned fault injection campaign. Table 2.5 lists the distribution of these modalities across multiple manufacturers. We observe that an average of 48% of all disengagements are initiated automatically by the system. Note that this measurement is biased by manufacturers like Mercedes-Benz and Waymo that report a larger number of disengagements.

Question 3: Dynamics of AV Disengagements

As suggested by Fig. 2.5, we expect that AV technology (including perception, decision,

Table 2.5: Distribution of disengagements across manufacturers (as percentages) categorized by modality.

Manufacturer	Automatic	Manual	Planned
Benz	47.11	52.89	0
Bosch	0	0	100
GMCruise	0	0	100
Nissan	54.2	45.8	0
Tesla	98.35	1.65	0
Volkswagen	100	0	0
Waymo	50.32	49.67	0

and control) gets tuned over time, resulting in decreasing DPMs. This hypothesis is true to varying degrees across manufacturers. In this section, we further assess its validity. In particular we look at 1. the temporal dynamics of DPMs (i.e., does DPM decrease with time?), and 2. the dynamics of DPM with the cumulative number of miles driven (i.e., does DPM decrease with more extensive testing?).

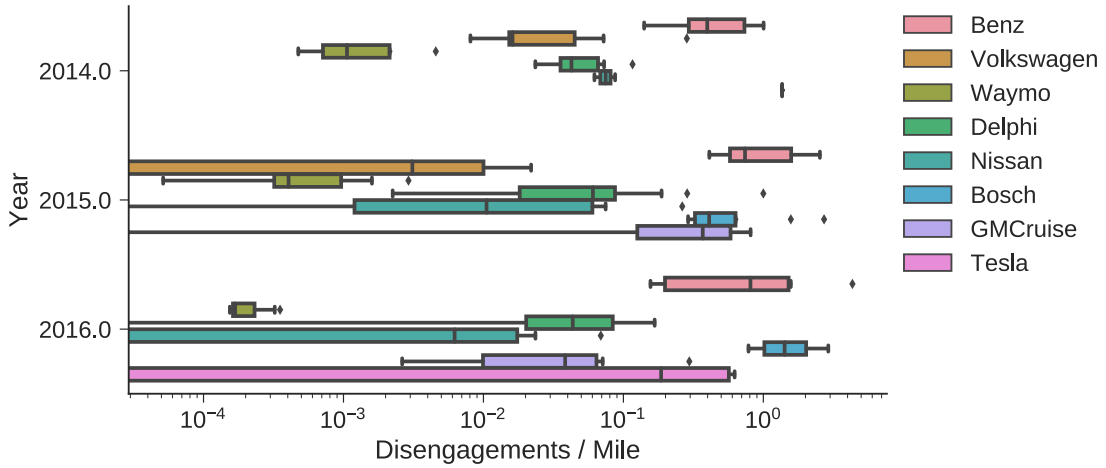


Figure 2.7: Time evolution (aggregated by year) of the distributions of DPMs per car across all manufacturers. The boxes show quartiles, notches show medians, and whiskers show max/mins.

Temporal Trends. Fig. 2.7 illustrates the temporal dynamics of the distribution of DPM per car across manufacturers aggregated per year. First, we observe that there is a distinct decreasing trend for the median DPM across most manufacturers. Some manufacturers, like Bosch that show an increase in median DPM per year claim that their disengagements result from planned fault injection experiments (see Table 2.5). In fact, some manufactur-

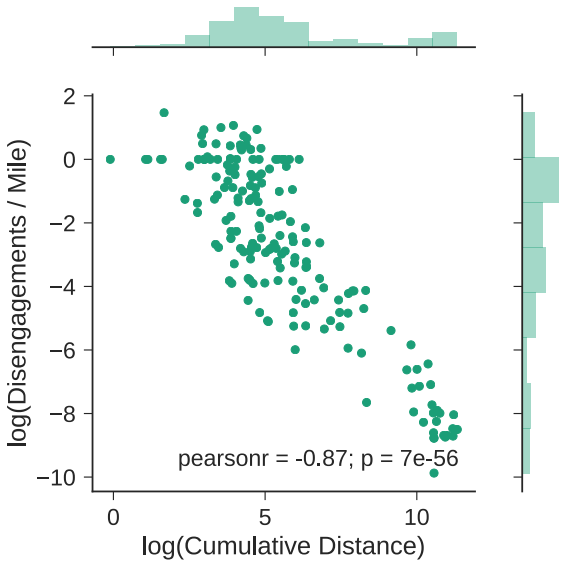


Figure 2.8: Linear statistical relationship between DPM per car and the cumulative number of autonomous miles.

ers show a decrease of as much as $10\times$ in median DPM across the three-year analysis window. Second, we see a significant increase in the variance of the DPM across cars over the period of interest. *This increase suggests that the median performance improves over time. However, the worst-case performance does not, since the variance relative to the median is large*. In fact, for some manufacturers, like Delphi, the 75^{th} percentile DPM across years changes by less than 50%. Waymo is an exception to this trend, demonstrating a nearly $8\times$ decrease in median DPM with a significant decrease in variance across the three years of measurement. Recall from Question 1 that Waymo is still not approaching the asymptote.

Trend with Cumulative Miles Driven. While the temporal trends are important, an alternative approach is to look at disengagements per mile as a function of miles driven. Since manufacturers do not all drive the same number of autonomous miles each month, this measure is a more equitable analysis of the AVs across manufacturers. Aggregating across all manufacturers, we observe that there is a strong negative correlation between DPM and cumulative miles driven (as shown in Fig. 2.8). We observe that the $log(DPM)$ and $log($cumulative autonomous miles$)$ are correlated with a Pearson coefficient of -0.87 (at a p-value of 7×10^{-56}). Fig. 2.9 shows this relationship across different manufacturers, with linear regression fit lines describing the trends mentioned above. That suggests that the manufacturers are continuously improving their ADSs, with some manufacturers making more headway than others (as represented by the slope of the fitted lines).

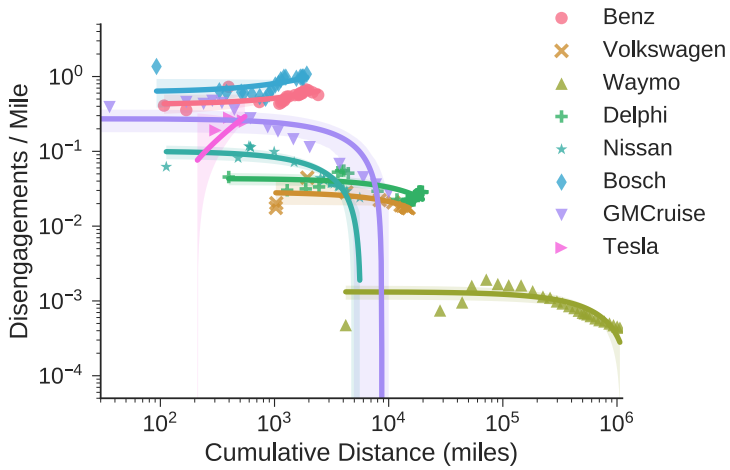


Figure 2.9: Evolution of DPM (per car) with the number of cumulative autonomous miles driven across all cars of that manufacturer. Lines represent a linear regression fit of each manufacturer's data.

Further, we observe that manufacturers with larger DPMs seem to make more significant improvements over the same number of miles driven; this suggests that some of the faults/problems fixed as a result of this testing represent the "low-hanging fruit."

While the temporal trends maybe more indicative of how actual users will drive these cars (i.e., the AVs will be used with a mix of idle and driving times), the trends with cumulative miles provide a more robust alternative for comparisons, wherein the miles driven are the only basis for comparison. Both show a decreasing trend the first shows an increasing variance; neither shows that any of the cars have approached a very low or zero DPM regime.

Question 4: Driver Alertness Level

The CA DMV defines *reaction time* as "the period of time elapsed from when the autonomous vehicle test driver was alerted of the technology failure, and the driver assumed manual control of the vehicle".[6] The case studies we presented in §2.2 highlight the need for the human driver in the AV to be alert and cognizant of the environment. The reaction times provide an understanding of how quickly an individual would react to a fault, and hence are essential for accident avoidance. Fig. 2.10 gives the distribution of test drivers' reaction times across all manufacturers. We observe an average $0.85\ s$ reaction time across all test vehicle drivers and all manufacturers. This observation is consistent with a similar observation made in [84]. Further, the distribution of reaction

[6]We assume the reaction times to be upper bounded where they are listed as ranges.

times is long-tailed. For example, Volkswagen reported at least one case with a near $4\ hr$ reaction time for a disengagement; we suspect that this is an incorrect measurement, but cannot confirm. Fig. 2.11 shows this long-tailed behavior with an Exponential-Weibull fit for the reported data for manufacturers other than Volkswagen.

Comparison to Human Alertness Levels. To understand whether that behavior is indeed representative of human alertness levels when driving, we compare those results with those presented in [85] for non-AVs. [85] found the reaction time for braking in test vehicles to be $0.82\ s$. This observation is consistent with our study. Further, [85] report that a driver's ownership of a vehicle (i.e., it is his or her own property) increased reaction time by approximately $0.27\ s$. Hence we assume $1.09\ s$ to be the average time for a human driver in a non-AV to respond any situation on the road. The observation implies that semi-AVs which are the most commonly deployed AVs on public streets) would require continued human supervision and alertness similar to human controlled non-AVs. Echoing the results of Question 3, that in turn suggests that the technology may not be mature enough to allow human drivers to be engaged in other activities, contrary to what is advertised.

Temporal Behavior of Reaction Time. We find that a driver's alertness decreases (i.e.,

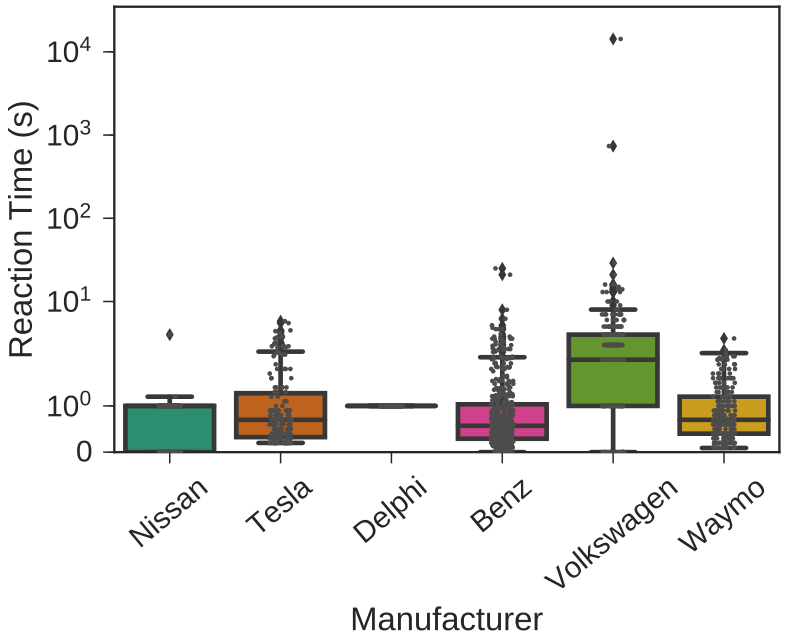
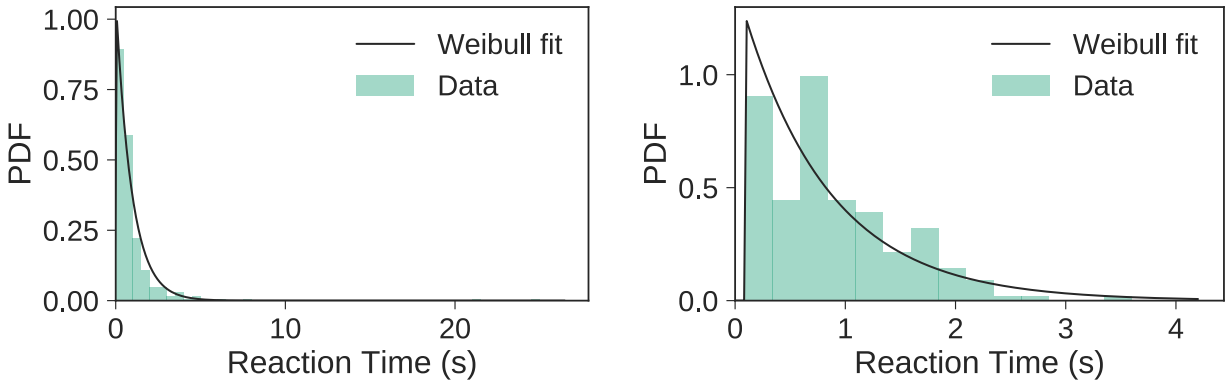


Figure 2.10: Distribution of reaction times for drivers in case of a disengagement across all manufacturers. The boxes show quartiles, notches show medians, and whiskers show max/mins.



(a) Mercedes-Benz

(b) Waymo

Figure 2.11: Distribution of reaction times for the Mercedes-Benz and Waymo.

reaction time increases) with the number of cumulative miles driven. At a 99% confidence level, we observe a positive correlation between the cumulative miles driven and the reaction times across manufacturers. For example, Waymo and Mercedes-Benz show a Pearson's correlation coefficient of 0.19 (at p-value $= 0.01$) and 0.11 (at p-value $= 0.007$), respectively. Taken together, that observation and the previous observation about decreasing DPM (described in §2.5.1) suggest that a driver's alertness decreases as the system's performance improves (i.e., DPM decreases).

Fault Detection Latency and Reaction Time. By definition, the reaction time does not include fault detection time. However, as our case studies show, the detection time is indeed part of the end-to-end time window in which the driver reacts to an adverse situation. For example, in both case studies presented in §2.2, the primary cause of the accident was the insufficient time left for the driver to make a decision after the fault was detected.

The drivers of AVs have to maintain the same level of alertness as when driving non-AVs. This suggests that the small size of the overall action window (detection time + reaction time) can make the reaction-time-based accidents a frequent failure mode with the widespread deployment of AVs. We also note that in planned test scenarios for AVs, drivers are required, trained, and paid to remain continuously attentive to the activities of the AV. Data for them might not generalize to regular users.

2.5.2 Analysis of AV Accident Reports

Question 5: Comparison to Human Drivers

Table 2.6: Summary of accidents reported by manufacturers.

Manufacturer	Accidents	Fraction of Total	DPA
Waymo	25	59.52	18
Delphi	1	2.38	572
Nissan	1	2.38	135
GMCruise	14	33.33	20
Uber ATC	1	2.38	–

DPA = Disengagements per accident.

To address this question, we define two additional measures: 1. accidents per mile (APM), and 2. disengagements per accident (DPA). We calculate the DPAs as shown in Table 2.6. As some of the accident reports were partially redacted by the CA DMV to obfuscate AV identification (e.g., the registration number or VIN number were removed), we cannot compute the APM per vehicle directly. We instead compute *accidents per mile* using the equation $APM = {}^{DPM}/_{DPA}$. Even though the number of accidents is small compared to the number of disengagements, we use [86] to test the statistical significance of our results. Our calculations for two out of the 4 manufacturers (i.e., Waymo and GMCruise) were made at $> 90\%$ significance.

Comparison of APMs across Manufacturers. We observe that there is great variability ($\sim 100\times$) in APMs across manufacturers (see Table 2.7). For example, Waymo is responsible for 59.52% of accidents reported (see Table 2.6), but has the lowest DPM (7.45×10^{-4}), the lowest DPA (18), and the lowest APM (4.14×10^{-5}). In contrast, GMCruise has a similar DPA (20) but performs $238\times$ worse in terms of DPM, and $214\times$ worse in terms of APM, as compared to Waymo (see Table 2.7). This suggests that there is significant variability across manufacturers in classifying the severity of disengagements, which again indicates the immaturity of the current AV technology. Also, the observed APM metric variability can be partially attributed to test drivers' proactive disengagement of the ADS (i.e., manual disengagement as presented in §2.5.1) to prevent accidents. We compare the accident rate of AVs with that of manual vehicles using data for [87, 88], which report that one accident is expected every $500,000$ miles (i.e., $APM = 2 \times 10^{-6}$). We find that compared to human drivers, AVs perform 15–$22\times$ worse (see Table 2.7) in terms of APM.[7]

When they are calculated using first principles (i.e., not using DPA as done before), for vehicles that can be identified in the accident reports, we observe a strong positive correlation between the number of accidents observed per mile and the number of au-

[7]Note that [87, 88] report only crashes on highways and freeways. However, AVs are required to report any crash on all types of roads.

Table 2.7: Reliability of AVs compared to human drivers.

Manufacturer	**Median DPM** (mile^{-1})	**Median APM** (mile^{-1})	**Rel. to HAPM**
Mercedes-Benz	0.565	–	–
Volkswagen	0.0181	–	–
Waymo	0.000745	4.140×10^{-5}	$20.7\times$
Delphi	0.0263	4.599×10^{-5}	$22.99\times$
Nissan	0.0413	3.057×10^{-4}	$15.285\times$
Bosch	0.811	–	–
GMCruise	0.177	8.843×10^{-3}	$4421.5\times$
Tesla	0.250	–	–

HAPM – Human APM.
Human APM $= 2 \times 10^{-6}\text{mile}^{-1}$ [87, 88].
Column 4 $=$ $^{\text{AV APM}}/_{\text{Human APM}}$.

tonomous miles driven (with a Pearson correlation coefficient of 0.98 at p-value < 0.01). Comparing that number to the trends in the DPM seen in Fig. 2.8, we see that there is a much stronger correlation of the APM with cumulative miles. This behavior might be indicative of the manufacturers' priority on fixing problems in their ADSs (i.e., they identify problems relating to accidents and fix them quickly).

Our analysis shows that for the same number of miles driven, for manufacturers that reported accidents, human-driven cars (non-AVs) are $15 - 4000\times$ *less likely to have an accident than AVs.*

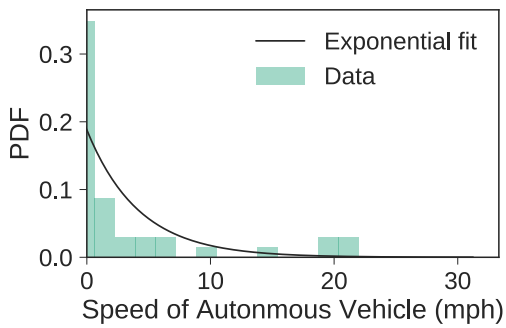
Collision Speeds and Locations. All the accidents reported in the dataset occurred at low speeds and in the vicinity of intersections on urban streets. Fig. 2.12 shows that more than 80% of the accidents occurred when the relative speed[8] of the colliding vehicles was less than $10\ mph$. In most of the cases in which the non-AV vehicle was determined to be at fault, the underlying cause can be attributed to the failure of the vehicle's driver to anticipate AV behavior. This observation points to the need for better understanding of the driving interactions and behaviors that drivers expect from other on-road vehicles. Most of the accidents were minor (either rear-end or side-swipe collisions), and no serious injuries were reported.

Our data show that better situational awareness needs to be provided by the ADSs (in particular the machine learning algorithms) to preemptively avoid accidents in a timely fashion.

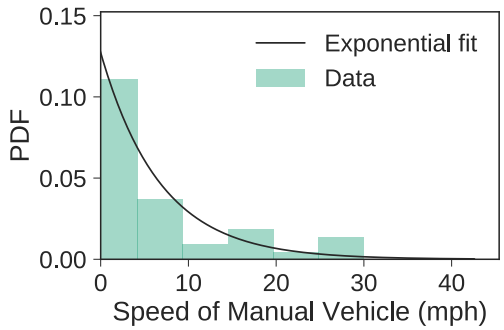
2.5.3 Discussion

Comparison to Other Safety-critical Autonomous Systems

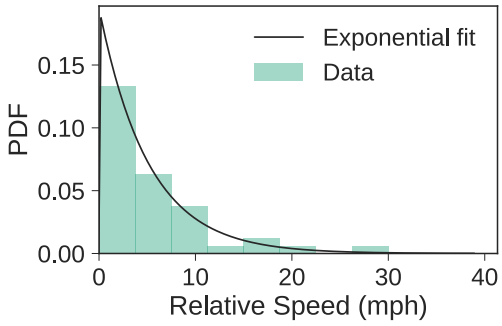
[8]The absolute difference between the speeds of the vehicles at the collision.



(a) Speed of AV



(b) Speed of MV



(c) Relative speed

Figure 2.12: Distribution of vehicular speeds for all reported accidents.

Airplanes [89] and surgical robots [90] are safety-critical semi-autonomous systems that have seen ubiquitous deployment, as well as a significant body of work characterizing and improving their resilience. We compare AVs to both of these systems in terms of the accidents per mission (APMi), to gauge the maturity of AVs vis-a-vis these systems. We define a *mission* as the continuous operation of the system of interest from the time of commencement to the end of the activity. For airplanes and cars, a mission is equivalent to one departure (i.e., trip), and for the surgical robot, a mission is equivalent to a surgical procedure.

We use data presented in [91] (9.8 accidents per $100,000$ departures for airplanes) and [92] (1043 accidents per $100,000$ procedures for surgical robots) as the baseline for comparison. We estimate the APMi of an AV by using data (pertaining to the average length of a vehicle ride on U.S. public roads for which there is a median of 10 miles per trip) presented in [93]. Using the APM metric computed earlier as shown in Table 2.8, we compute APMi as APM $\times$ length of the average trip. Our analysis shows that AVs do surprisingly well per mission. *Compared to airplanes (which utilize sophisticated resilience*

Table 2.8: Reliability of AVs compared to other safety-critical autonomous systems.

Manufacturer	**APMi**	**Aviation Industry** $^{\text{APMi}}/_{\text{Airline APM}}$ [91]	**Surgical Robotics** $^{\text{APMi}}/_{\text{SR APM}}$ [92]
Waymo	4.140×10^{-4}	4.22	0.0398
Delphi	4.599×10^{-4}	4.69	0.0442
Nissan	3.057×10^{-3}	31.19	0.293
GMCruise	8.843×10^{-2}	902.34	8.502

APMi = Accidents per mission for an AV
Airline APM = 9.8×10^{-5}
Surgical Robot (SR) APM = 1.04×10^{-2}

models and techniques), AVs are merely $4.22\times$ *worse, and are* $2.5\times$ *better than surgical robots (see Table 2.8).*

However, if all cars are replaced by AVs in the future, the AVs will make ~ 96 billion trips per year [94], compared to the 9.6 million trips for airlines. This means that AVs will make $10,000\times$ more trips than airlines, leading to a higher number of accidents per year than for airplanes. Further, the average length of a mission in terms of time and miles covered is significantly different for airplanes and AVs. Hence a holistic comparison across these systems would need to consider operational time per mission, as well as account for competing failures across concurrent deployments of these systems.

Traditional Reliability Metrics

While we have made an approximate comparison above, the more traditional and accurate method for comparing the resilience of AVs with that of airplanes (which are also highly automated systems) is via operational hours to failure. That metric, however, is unavailable for cars, since we do not have information about the idle time for these vehicles or its distribution. We propose an alternative metric based on the number of miles driven to disengagement/accident. This metric will be available across transportation systems.

To directly obtain this measure, there needs to be a small change in the data collection by the DMV: manufacturers and the DMV should collect data on miles between disengagements per vehicle to enable the computation of the metrics.

2.6 THREATS TO VALIDITY

An empirical study like ours is subject to vagaries arising from heterogeneous data collection systems (e.g., the inclusion or exclusion of data points, or the disparate in-

formation content across data formats), thus hampering the ability to draw generalized conclusions. Dealing with such issues is not uncommon in the realm of system reliability assessment. We assert the need for replication studies to verify our conclusions across other datasets. We now discuss potential threats to validity that are specifically related to our study.

Construct Validity implies that variables associated with the study are measured correctly, i.e., that the measurements are constructed in accordance with the theoretical foundations of the area. We have discussed construct validity in §2.5.3.

Internal Validity implies that there are no systematic errors and biases. We studied the datasets available from 12 different manufacturers and only reported generalized trends in order to eliminate any biases and micro-observations (observations with low statistical significance) that might be artifacts of bad logging or biases from the manufactures in reporting the disengagements and accidents. For example:

- *Data underreporting*: In order to obtain an AV testing permit, companies are legally required to catalogue and submit to the DMV reports of all disengagements and accidents that 1. pertained to technology failures and safe operation of the AVs, and 2. required the AV test driver to disengage the autonomous mode and take immediate manual control of the vehicle. The interpretation of "safe" operation and technology "failure" can vary across manufacturers, leading to underreporting. Further, regulatory oversight and enforcement of regulations are difficult and may result in underreporting. Given the available data, we cannot accurately estimate the scale of underreporting, and hence refrain from drawing any such conclusions.
- *Not all miles are equivalent*: One manufacturer may hold the tests of its AVs in more challenging environments than others do, e.g., at night or during bad weather. Not all manufacturers report environmental conditions during tests. Where available, we report the testing conditions and disengagements caused by environmental factors (see "Environment" in Fig. 2.6).
- *Validity of fault tags and failure categories*: There is no consistent data format for the provided disengagement/accident reports across manufacturers. Our NLP framework for tagging and categorization may lead to systematic errors; therefore, the dictionaries were verified manually by the authors to ensure their correctness. We explicitly labeled data points as "Unknown-T/C" when there was uncertainty in the tags and categories given by the NLP framework.

External validity concerns the extent to which a study can be generalized to other systems or datasets. To the best of our knowledge, the CA DMV dataset is the only publicly available dataset pertaining to AV failures. Until we work with manufacturers on propri-

etary data (which might not be disclosed publicly), we cannot comment on the general external validity of the techniques presented here.

2.7 RELATED WORK

The majority of the prior research into AV systems focuses on the functionality of vehicle guidance systems. Numerous demonstrations of end-to-end computing systems for autonomous vehicles have recently been done (e.g., [59–61, 95, 96, 62–64]). The currently accepted practice for vehicular safety, based on the ISO 26262 safety standard [97], is to consider human drivers to have ultimate responsibility for safety. That is the basis for most AV testing programs on public roads, which require a safety driver to be in the vehicle to monitor the vehicle. This driver is expected to intervene if a system failure occurs that leads to a disengagement or accident; indeed, we observe several such incidents in the CA DMV datatsets. In such a scenario, safety considerations for the AV are driven by 1. the AV's ability to alert the driver in case of failure, 2. the driver's ability to recognize the abilities of the AV and the limits of the system, 3. the AV's ability to anticipate the behavior of other road users who might not always conform to the rules, and 4. the other road user's ability to anticipate the behavior of the AV [98, 99]. How this will be handled in autonomous vehicles remains an open question [100]. Safety is also emphasized in a number of publications, including [101, 102]. Waymo has published a report on the safety precautions considered for their AVs [20].[9]

[86] provides a model to estimate the number of miles AVs have to be driven to demonstrate their reliability with statistical confidence. [81, 84] provide summary statistics (e.g., driver reaction times and AV speed in accident scenarios) from tabulated data in the DMV dataset. Our approach uses an STPA based ontology and NLP techniques (which in itself are novel contributions of this work) to parse a significant amount of unstructured data presented as natural text.

[75] use fault injection to evaluate the fault tolerance of deep neural networks (DNN: used primarily in the *Sensor Fusion & Environmental Information Processing* step shown in Fig. 2.3), analyze the DNN's results, and propose techniques to safeguard DNNs from single-event upsets. In contrast, we present an analysis of the entire control system of the AV, of which DNNs are a small part.

Other related work has focused on safety and reliability of AVs as they apply to legal

[9] For their trained drivers, Waymo claimed there was 1 accident for 2.3 million miles; we cannot substantiate that.

(e.g., [67, 68]) and regulatory barriers (e.g., [69–72]) for AV deployment and implementation.

Security and privacy measures to encompass system-level attacks and failures of AVs have also been studied [103, 104].

2.8 CONCLUSIONS AND FUTURE WORK

A steady march toward the use of AVs is clearly under way. The reliability and safety challenges of fully-autonomous vehicles (Level 4 & 5, currently under development) and today's semi-AVs are significant and underestimated. We therefore draw the following conclusions to frame our future research and draw the attention of other reliability researchers.

- There is ongoing research on the verification and validation of the safety properties of individual system components (e.g., the control, communication, and mechanical system components) using the STAMP framework [101]. However, our study shows there is a need for rigorous theoretical models (like STPA models) for evaluating AV technologies.
- The machine learning systems responsible for perception and control need further research and assessment under fault conditions via stochastic modeling and fault injection to augment data collection.
- In reality, there is a strong possibility that both AVs and semi-AVs will co-exist with non-AVs (with human drivers completely in charge) within several years. Therefore the urgency of joint study driven by data and models needs to be emphasized.

CHAPTER 3: AV: DOMAIN-GUIDED ML FOR RAPID ASSESSMENT

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

3.1 INTRODUCTION

Autonomous vehicles (AVs) are complex systems that use artificial intelligence (AI) and machine learning (ML) to integrate mechanical, electronic, and computing technologies to make real-time driving decisions. AI enables AVs to navigate through complex environments while maintaining a *safety envelope* [12, 13] that is continuously measured and quantified by onboard sensors (e.g., camera, LiDAR, RADAR) [14–16]. Clearly, the safety and resilience of AVs are of significant concern, as exemplified by several headline-making AV crashes [3, 2], as well as prior work characterizing AV resilience during road tests [6]. Hence there is a compelling need for a comprehensive assessment of AV technology.

AV development today involves verification [102, 105–107], validation [108], and testing [109, 110] as well as other forms of assessment throughout the life cycle. However, assessment of these systems in realistic execution environments, especially because of the occurrence of random faults, has been challenging. Fault injection (FI) is a well-established method for testing the resilience and error-handling capabilities of computing and cyber-physical systems [111] under faults. FI-based assessment of AVs presents a unique challenge not only because of AV's complexity but also because of the centrality of AI in a free-flowing operational environment [112]. Also, AVs represent a complex integration of software [113] and hardware technologies [114] that have been shown to be vulnerable to hardware and software errors (e.g., SEUs [115, 116], *Heisenbugs* [117]). Future trends of increasing code complexity and shrinking feature sizes will only exacerbate the problem.

This chapter presents *DriveFI*, an intelligent FI framework for AVs that addresses the above challenge by identifying hazardous situations that can lead to collisions and accidents. DriveFI includes (a) an FI engine that can modify the software and hardware states of an autonomous driving system (ADS) to simulate the occurrence of faults, and (b) an ML-based fault selection engine, which we call *Bayesian fault injection*, that can find the situations and faults that are most likely to lead to violations of safety conditions. In contrast, traditional FI techniques [111] often do not focus on safety violations, and in practice have low manifestation rates and require enormous amounts of time under test [118, 75]. Note that given a fault model, DriveFI can also perform random FI to obtain a baseline.

Contributions. DriveFI's Bayesian FI framework is able to find safety-critical situations and faults through causal and counter-factual reasoning about the behavior of the ADS under a fault. It does so by (a) *integrating domain knowledge* in the form of vehicle kinematics and AV architecture, (b) *modeling safety* based on lateral and longitudinal stopping distance, and (c) using *realistic fault models* to mimic soft errors and software errors. Items (a), (b), and (c) are integrated into a *Bayesian network* (BN). BNs provide a favorable formalism in which to model the propagation of faults across AV system components with an interpretable model. The model, together with fault injection results, can be used to design and assess the safety of AVs. Further, BNs enable rapid probabilistic inference, which allows DriveFI to quickly find safety-critical faults. The Bayesian FI framework can be extended to other safety-critical systems (e.g., surgical robots). The framework requires specification of the safety constraints and the system software architecture to model causal relationship between the system sub-components. We demonstrate the capabilities and generality of this approach on two industry-grade, level-4 ADSs [119]: DriveAV [14] (a proprietary ADS from NVIDIA) and Apollo 3.0 [15] (an open-source ADS from Baidu).

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

- Using fault model (b) we compiled a list of 98,400 faults. An exhaustive evaluation of all 98,400 faults in our simulated driving scenarios would have taken 615 days. In comparison, our Bayesian FI was able to find 561 faults that maximally impact AV safety in less than 4 hours. Thus, Bayesian FI achieves $3690\times$ acceleration. Two cases found by Bayesian FI are described in §3.2.4; one, in particular, mimics the Tesla vehicle crash [3].
- Bayesian FI is able to find critical faults and scenes that led to safety hazards. (a) Out

of the 561 identified faults, 460 manifested as safety hazards. (b) These 460 faults were found to be associated with 68 safety-critical scenes[1] (out of 7200 scenes).

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

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

Putting DriveFI in Perspective. Early work studied the safety of AVs using system-theoretic approaches [19, 101]. More recent studies have focused on the resilience of constituent modules of an ADS (described in §3.4), e.g., [75, 120–122]. Another line of work [8, 123] has used FI to study sensor-related resilience in AVs. In contrast to DriveFI, none of the prior approaches have considered the resilience of modern end-to-end AI-driven systems that use industry-grade ADSs to mine faults that lead to safety hazards.

3.2 APPROACH OVERVIEW

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

3.2.1 Autonomous Driving System

Fig. 3.1 illustrates the basic control architecture of an AV (henceforth also referred to as *Ego Vehicle*, EV). It consists of mechanical components and actuators that are controlled by an *ADS*, which represents the computational (hardware and software) component of the AV. At every instant in time, t, the ADS system takes input from sensors $\mathbf{I_t}$ (e.g., cameras, LiDAR, GPS), takes inertial measurements $\mathbf{M_t}$ from the mechanical components (e.g., velocity v_t, acceleration a_t), and infers actuation commands $\mathbf{A_t}$ (e.g., throttle ζ, brake b, steering angle ϕ). For clarity, we further subdivide the ADS into two components: (a) an ML module (responsible for perception and planning) that takes as inputs $\mathbf{I_t}$ and $\mathbf{M_t}$ and produces raw-actuation commands $\mathbf{U_{A,t}}$, and (b) a PID controller [124] that is responsible for smoothing the output $\mathbf{U_{A,t}}$ to produce $\mathbf{A_t}$. The PID controller ensures that the AV does

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

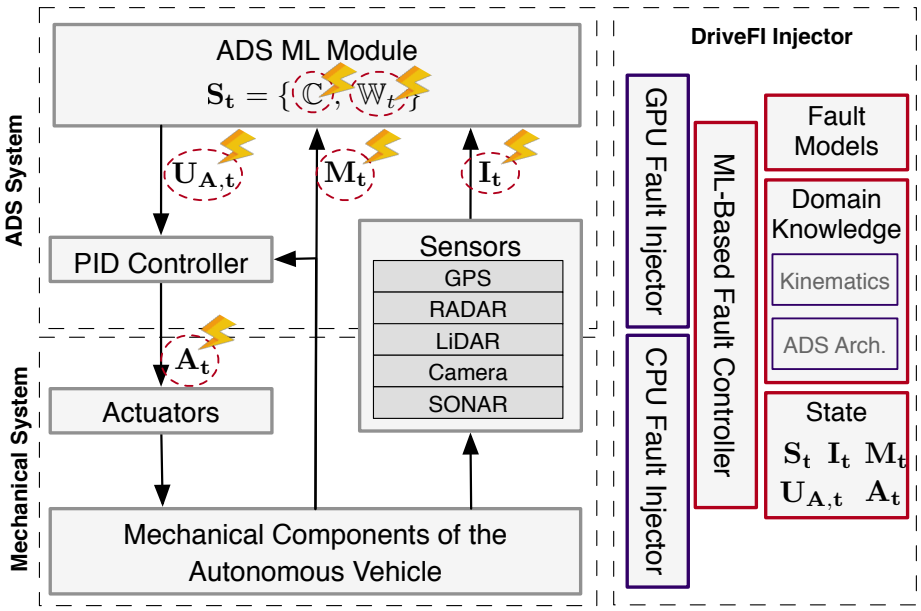


Figure 3.1: A high-level overview of the AV's autonomous and mechanical systems, and its interaction with DriveFI.

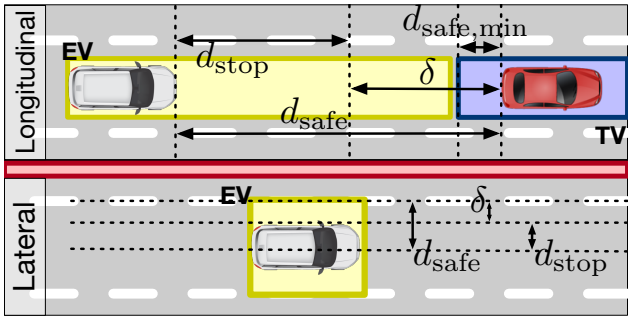


Figure 3.2: Definition of d_{stop}, d_{safe}, and δ for lateral and longitudinal movement of the car. Non-AV vehicles are labeled as *target vehicles* (TV).

not make any sudden changes in $\mathbf{A_t}$. The ADS ML module has an instantaneous state $\mathbf{S_t}$ that consists of configuration parameters $\mathbb{C}$ (e.g., neural network weights to perceive input camera data) and a *world model* $\mathbb{W}_t$, which maintains and tracks the trajectories of all static objects (e.g., lane markings) and dynamic objects (e.g., other vehicles) perceived by the ADS.

3.2.2 Safety

We define the instantaneous safety criteria of an AV in terms of the longitudinal (i.e., direction of motion of the vehicle) and lateral (i.e., perpendicular to the direction of the

vehicle motion) Cartesian-distance travelled by the AV (see Fig. 3.2). Those criteria form a "primal" definition of safety based on collision avoidance, which can be extended with other notions of safety, e.g., using traffic rules. The extended notions of safety are not considered in this chapter, as they can be nuanced based on the laws of the geographic regions in which they are applied.

Definition 3.1. The *stopping distance* d_{stop} is defined as the maximum distance the vehicle will travel before coming to a complete stop while the maximum comfortable deceleration a_{max} is being applied.

Definition 3.2. The *safety envelope* d_{safe} [12, 13] of an AV is defined as the maximum distance an AV can travel without colliding with any static or dynamic object.

A safety envelope is used to ensure (through constraints on $\mathbf{U_{A,t}}$) that the vehicle trajectory is collision-free. Production ADSs use techniques such as those in [125, 126] to estimate vehicle and object trajectories, thereby computing d_{safe} whenever an actuation command is sent to the mechanical components of the vehicle. These ADSs generally set a minimum value of d_{safe} (i.e., $d_{\text{safe,min}}$) to ensure that a human passenger is never uncomfortable about approaching obstacles.

Definition 3.3. The *safety potential* δ is defined as $\delta = d_{\text{safe}} - d_{\text{stop}}$. An AV is defined to be in a *safe state* when $\delta > 0$ in both lateral and longitudinal directions.[2]

3.2.3 Fault Injection

The goal of DriveFI is to test ADSs in the presence of faults to identify hazardous situations that can lead to accidents (e.g., loss of property or life). To accomplish that goal, DriveFI includes (a) an FI engine that can modify the software and hardware states of the ADS to simulate the occurrence of faults, and (b) an ML-based fault selection engine that can find the faults and scenes that are most likely to lead to violations of safety conditions and, hence, can be used to guide the fault injection. Taken together, these components of DriveFI can identify hazardous situations that lead to accidents similar to the Tesla crash described later in this section.

Fault Model. We assume that faults injected in DriveFI can corrupt GPU architectural state. Memory and caches (of both the CPUs and GPUs) are assumed to be protected with SECDED codes. Each injected fault is characterized by its location (in this case, its

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

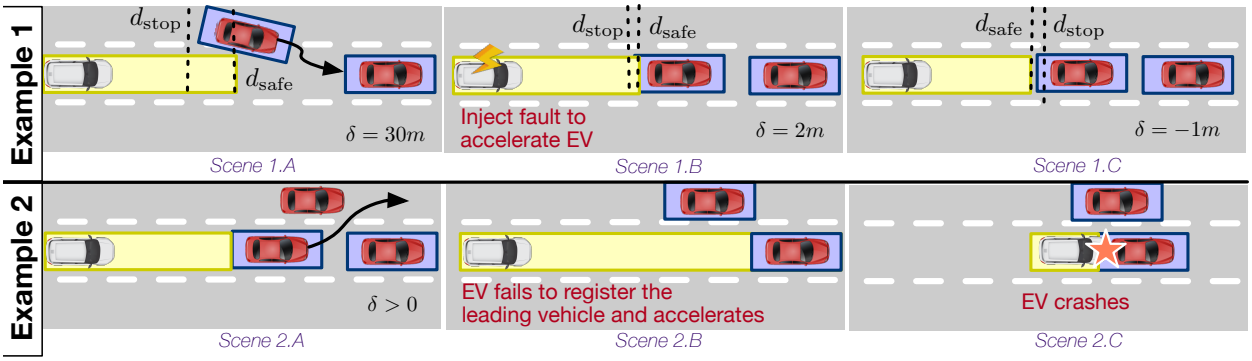


Figure 3.3: Example scenarios: (1) Targeted FI leads to hazardous conditions; (2) Real-world example with Tesla Autopilot that is similar to injected faults.

dynamic instruction count) and the injected value. The faults injected into the architectural states of these processors can manifest as *errors* in the inputs, outputs, and internal state of the ADS modules described above (i.e., $\mathbf{I_t}$, $\mathbf{M_t}$, $\mathbf{S_t}$, $\mathbf{U_{A,t}}$ and $\mathbf{A_t}$). DriveFI can directly inject errors into ADS outputs by corrupting the variables that store ADS outputs. ADS software input/output variables are ultimately stored in different levels of storage hierarchies, e.g., registers or caches. Single- or multiple-bit faults cause corruption of variables when not masked in hardware [1]. Hence, faults are being injected into these memory units, but the variables are corrupted to emulate the faults. Therefore, our fault injectors target each element in the internal ADS software state (S_t), sensor inputs (I_t), vehicle inertial measurements (M_t), and actuation commands (U_t, A_t), as shown in Fig. Fig. 3.10. We define any error that causes safety issues for the AV as *hazardous*. For simplicity and clarity, in the remainder of the chapter, we refer to both injected faults and errors as *faults*.

To build a baseline for the ML-based targeted injections, we used DriveFI to perform random injections into the GPU architectural state and ADS module outputs for two production ADS systems from NVIDIA and Baidu. In contrast to prior work [75, 122], which has reported significant SDC rates (as high as 20%) for the constituent deep-learning models (ConvNets that deal with perception: object recognition and tracking) of the ADS system, we observed that random injections rarely cause hazardous errors. These faults are masked because of the natural resilience of the ADS stack, i.e., (a) for production ADS systems that make real-time inferences at 60–100 Hz, transient faults have little chance to propagate to actuators before a new system state is recalculated; (b) the ADS system architecture is inherently resilient, as it uses algorithms like extended Kalman filtering [127] (for sensor fusion) and PID control (for output smoothing); and (c) not all driving scenes/frames are hazardous even under faults. Environmental conditions, such

as the presence of other objects on the streets, are fundamental in defining the safety envelope.

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

$$\mathbf{F}_{\text{crit}} = \left\{ f : \delta > 0 \wedge \hat{\delta}_{\text{do}(f)} \leq 0 \right\}. \tag{3.1}$$

The solution to that problem requires causal and counter-factual reasoning about the behavior of the ADS under a fault. DriveFI performs that reasoning by modeling the ADS system using a *Bayesian network* (BN; shown in Fig. 3.4), which can capture causal relationships [128].

The BN describes statistical relationships shown by black arrows between the variables $\mathbb{W}_t$, $\mathbf{M_t}$, $\mathbf{U_{A,t}}$, and $\mathbf{A_t}$ at a time t, as well as relationships shown by red arrows between the variables over time. The topology of the BN is derived from the architecture of the ADS system. For example, Fig. 3.4 has the same graphical structure as Fig. 3.1. DriveFI uses the BN to calculate the maximum likelihood estimate (MLE)[3] of the value $\hat{\mathbf{M}}_{\mathbf{t+1}}$ and then uses the MLE value to calculate $\hat{\delta}_{\text{do}(f)}$ based on the kinematic model of the AV described later in §3.3. We use probabilistic inference over the posterior distribution of the BN to calculate

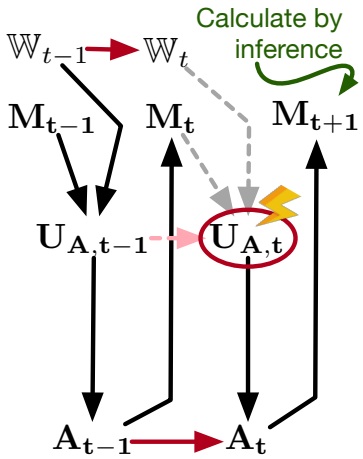


Figure 3.4: Bayesian FI.

$$\hat{\mathbf{M}}_{\mathbf{t+1}} = \arg\max_{\mathbf{m}} \Pr\left[\mathbf{M_{t+1}} = \mathbf{m} \mid \text{do}(f)\right]. \tag{3.2}$$

The $\text{do}(\cdot)$ notation is based on the *do-calculus* defined in [128]. It marks an FI action as an intervention in the BN model. It replaces certain probabilities with constants and removes statistical conditional dependencies that are a target of the intervention (i.e., dashed lines in Fig. 3.4), but preserves all other statistical dependencies. We call this notion of counterfactual reasoning about the importance of a fault in performing targeted injections *Bayesian Fault Injection*.

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

3.2.4 Case Studies

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

Example 1: Hazardous Error. Fig. 3.3 shows an example driving scenario in which a fault was injected into an ADS through corruption of the throttle command (which was changed from 0.2 to 0.6). The injected error led to an accident. We assume that (a) the ADS is running perception, planning, and control inference at 30 Hz, and (b) all vehicles are running on a highway with a velocity of 33.5 m/s, which is roughly the speed limit on U.S. freeways. In Scene 1A, the Ego vehicle (EV) was accelerating; however, target vehicle TV#1, operated by a human, initiated a lane change procedure, which decreased the safety potential $delta$ from 20 m to 2 m as shown in "Scene 1B." At that point, the Bayesian fault injector injects a fault into the throttle command, causing the vehicle to accelerate. The increase in acceleration caused the EV to become unsafe ($\delta < 0$), as shown in "Scene 1C." The EV velocity is high enough that braking, even with $a_{\max}$, is not able to prevent an accident. This example shows that one needs a smart FI mechanism (such as our Bayesian-based injector) that is able to inject a fault at a precise time instant based on a run-time measurement of the safety potential to maximize the stress on the ADS and cause the EV to crash. As we argue in later sections, it is impractical (or highly difficult) to achieve the same objective using random FI.

Example 2: Real-World Crash. Fig. Fig. 3.3 shows a real-world example of a fatal accident that was shown to have been caused by a problem in Tesla Autopilot [3]. In Scene 2A, the EV followed the lead vehicle (TV#1). A few seconds later, TV#1 changed lanes (shown as Scene 2B); at that point, Autopilot decided to accelerate in order to match the allowed highway speed. However, TV#1 was behind another vehicle (TV#2), and the EV had no knowledge of TV#2; it was too late for the EV to recognize TV#2 and slow down in time to avoid an accident. While this crash was attributed to a design problem (i.e., delayed recognition) in the perception subsystem of the ADS, one can imagine that a runtime fault (that delays perception of an object) could lead to the same fatal outcome. As we show later, our Bayesian-based fault injector is able to recreate such scenarios.

3.3 BAYESIAN FAULT INJECTION

Here we describe in detail the formulation of the *Bayesian Fault Injection* approach.

3.3.1 Kinematics-Based Model of Safety

Consider an EV moving in two-dimensional space as shown in Fig. 3.5. The vehicle at

time t has an instantaneous position (x_t, y_t), speed v_t, heading θ_t, and steering angle ϕ_t. The equations of motion for the vehicle are

$$dx_t/dt = v_t \cos\theta_t; \; dy_t/dt = v_t \sin\theta_t; \; d\theta_t/dt = (v_t \tan\phi_t)/L, \tag{3.3}$$

where L is the distance between the wheels of the EV [129]. Here v_t and ϕ_t are determined by the control model for the EV. In our case, v_t is defined based on the output of the ADS $\mathbf{A_t}$, i.e., $v_t = f(\zeta_t, b_t, \phi_t)$.

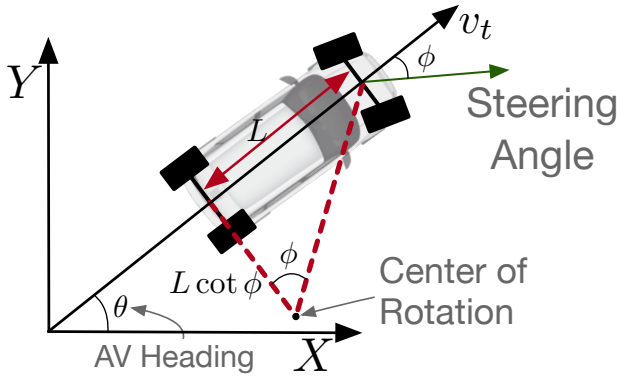


Figure 3.5: Orientation of the EV when in motion.

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

We can compute the maximum stopping distance d_{stop} from eq. (3.3) by first computing the time t_{stop} taken to bring the vehicle to a complete halt, i.e.,

$$\left.\frac{dx_t}{dt}\right|_{t=t_{\text{stop}}} = 0 \text{ and } \left.\frac{dy_t}{dt}\right|_{t=t_{\text{stop}}} = 0. \tag{3.4}$$

d_{stop} is then calculated as $[x_{t_{\text{stop}}} - x_0, y_{t_{\text{stop}}} - y_0]^T$, where (x_0, y_0) is the position of the EV at the beginning of the maneuver. Closed-form solutions to the system of differential equations eqs. (3.3) and (3.4) are intractable for arbitrary control procedures (i.e., v_t and ϕ_t) and have to be solved by iterative numerical solution methods like the *Runge-Kutta methods* [130].

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

$$\frac{dv_t}{dt} = -a_{\max} \text{ and } \frac{d\phi_t}{dt} = 0. \tag{3.5}$$

That corresponds to the deceleration of the EV with the maximum deceleration to come to a halt. eq. (3.5) reduces eq. (3.3) to

$$d^2x_t/dt^2 = -a_{\max} \sin\theta_t (d\theta_t/dt) \tag{3.6a}$$

$$d^2y_t/dt^2 = -a_{\max} \cos\theta_t (d\theta_t/dt) \tag{3.6b}$$

$$\frac{d\theta_t}{dt} = \frac{(\sqrt{(dx_t/dt)^2 + (dy_t/dt)^2})}{L} \tan\phi_0, \tag{3.6c}$$

where ϕ_0 is the steering angle of the car at the beginning of the maneuver. DriveFI uses the system of equations defined in eqs. (3.4) and (3.6) to find d_{stop}. We use the shorthand $\mathcal{P}$ to denote the procedure (iterative numerical integration) used to compute

$$d_{\text{stop}} = \mathcal{P}(a_{\max}, v_0, \theta_0, \phi_0, x_0, y_0) \tag{3.7}$$

from the above equations and the initial kinematic state of the EV (i.e., v_0, θ_0, ϕ_0, x_0, y_0) at the start of the maneuver.

Recall from §3.2 that $\delta = d_{\text{safe}} - d_{\text{stop}}$ and that $\delta > 0$ defines the safety of the EV. The d_{safe} value is assumed to be computed directly from the sensors of the EV. It is the distance to the closest object (static or dynamic) in the longitudinal or lateral path of the EV. As a result, d_{safe} changes with time, and it is updated at the sensor's (e.g., LiDAR's or camera's) refresh rate. We include the boundaries of the lane in which the EV is travelling (henceforth referred to as the *Ego lane*) as a static object to be used in d_{safe} computations to ensure that we capture lane violations as a safety hazard.

Discretization. We convert the problem of solving eq. (3.7) from one that uses continuous time to one that uses a discrete notion of time. Discrete time is a natural fit for the ADS, as the control decisions are made at discrete steps that correspond to the sensors' sampling frequencies. Hence we convert time t to a discrete number $k \in \mathbb{N}$ such that $t = k\Delta t$, where Δt is the period of the sensor with the smallest sampling frequency. In the case of the DriveFI injector in DriveWorks and Apollo, that is 7.5 Hz. However, our methodology is frequency-agnostic.

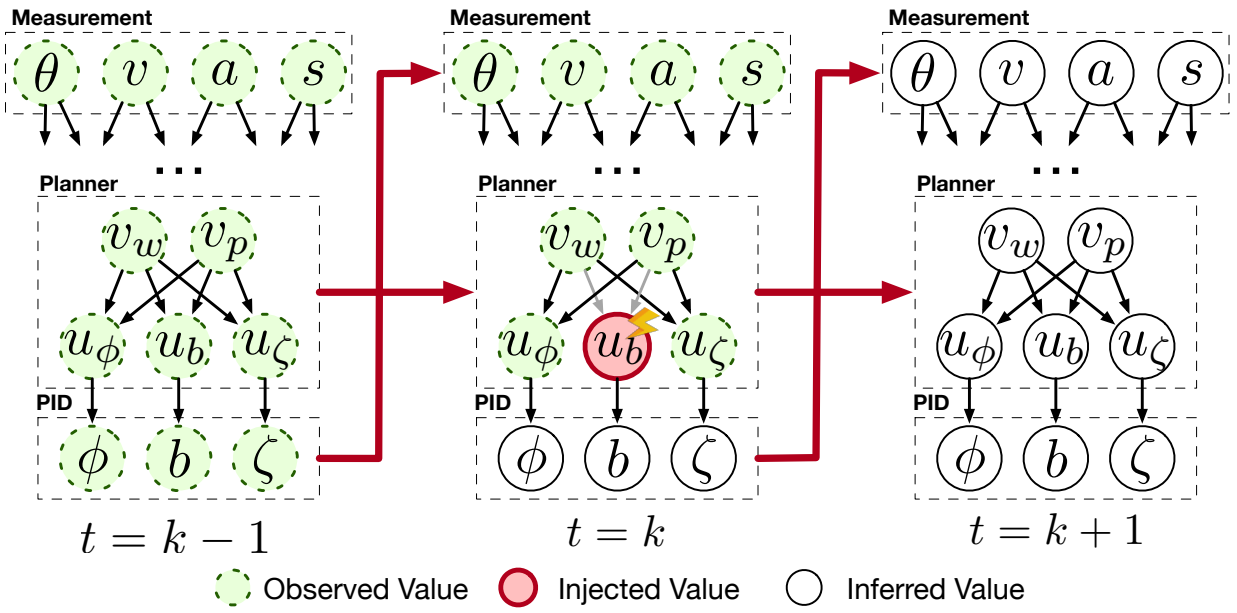


Figure 3.6: 3-Temporal Bayesian Network modeling the ADS.

3.3.2 ML Model

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

The Model. Consider a situation in which a fault is injected into the EV's ADS at time point k. We want to estimate the value of d_{stop} at time $k+1$ when the (corrupted) actuation commands of the previous time step have been acted upon. As we showed in the previous section, we can do so using eq. (3.7). However, that would require knowing the values x_{k+1}, y_{k+1}, v_{k+1}, θ_{k+1}, and ϕ_{k+1} as the initial conditions to start the emergency stop maneuver. DriveFI estimates those values based on a maximum likelihood estimation over the posterior distribution of a probabilistic model that captures the components of the ADS.

DriveFI uses a Dynamic Bayesian Network (DBN) [131], specifically a 3-Temporal Bayesian Network (TBN), i.e., a DBN unfolded thrice, to model x_{k+1}, y_{k+1}, v_{k+1}, θ_{k+1}, and ϕ_{k+1}. This model is illustrated in Fig. 3.6. The core idea of DBNs is to model each point in time with a static BN and to add temporal links from one time-slice to the next (as shown by red

arrows in Fig. 3.6). Usually all the time-points have identical BN topologies and hyper-parameter settings. BNs are directed acyclic graphs in which nodes represent random variables and arcs represent the causal connections among the variables [132]. Henceforthm we will refer to each random variable in the BN is henceforth referred to as a *node* to avoid confusion with the ADS variables. Each node x is associated with a probability table that provides conditional probability distributions (CPDs; $\Pr(x \mid \pi(x))$ of a node's possible value given the value of its parent nodes $\pi(x)$.

The 3-TBN model (see Fig. 3.6) is constructed based on the topological structure shown in Fig. 3.1. A detailed version of this figure for the Apollo and DriveFI ADSs is described in §3.4 and shown in Fig. 3.8. The variables in each of the ADS modules are connected in a parent-child fashion that reflects the data-flow in Fig. 3.8. For example, the edges between u_ζ and ζ (in Fig. 3.6) represent the CPD $\Pr(\zeta \mid u_\zeta)$. This is an approximation of the PID control for ζ. Similarly, other components of the ADS are modeled based on their input and output variables. We assume that the nodes in the 3-TBN are described by a CPD that has the functional form given by eq. (3.8).

$$\Pr(x \mid \pi(x)) = \mathcal{N}(\mu_{\mathbf{x}}^{\mathbf{T}} \pi(x), \sigma_x) \tag{3.8}$$

In eq. (3.8), $\mathcal{N}$ is the normal distribution with parameters $\mu_{\mathbf{x}}$ and σ_x (for each node x in the network). That particular form of $\Pr(x \mid \pi(x))$ is chosen because (a) it has numerical stability at small probability values, which are common when dealing with rare events like faults, and (b) it simplifies the algorithm required to train the 3-TBN.

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

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

$$\hat{v}_{k+1} = \arg\max_{v} \Pr\left(v_{k+1} = v \mid \mathrm{do}(f), \mathbf{O}_{\mathbf{k}}^{(\mathbf{f})}\right). \tag{3.9}$$

Given that we can execute a simulation of the EV under non-fault conditions, all variables that are not children of the injected variable can be observed to have values from the correct run. These "golden" observations are labeled $\mathbf{O}_{\mathbf{k}}^{(\mathbf{f})}$. eq. (3.9) is solved by first estimating the posterior distribution of v_{k+1} by using *Markov Chain Monte Carlo* methods [131] and then estimating the most likely value of v_{k+1}. A similar procedure can be

used to compute $\hat{\theta}_{k+1}$ and $\hat{\phi}_{k+1}$. The values of $\hat{x}_{k+1}$ and $\hat{y}_{k+1}$ can then be computed using time-discretized versions of eq. (3.3). Finally, from eq. (3.7), we get

$$\hat{d_{\text{stop}}} = \mathcal{P}\left(a_{\text{max}}, \hat{x}_{k+1}, \hat{y}_{k+1}, \hat{v}_{k+1}, \hat{\theta}_{k+1}, \hat{\phi}_{k+1}\right) \tag{3.10}$$

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

$$\Pr(\mathbb{X}_{k-1}, \mathbb{X}_k, \mathbb{X}_{k+1}) = \frac{1}{Z} \prod_{x \in \mathbb{X}_{k-1} \cup \mathbb{X}_k \cup \mathbb{X}_{k+1}} \Pr(x | \pi(x)) \tag{3.11}$$

In eq. (3.11), Z is the partition function that normalizes P to be a probability distribution. We use the *Expectation-Maximization algorithm* [133] to compute

$$\hat{\mu}, \hat{\sigma} = \arg\max_{\mu,\sigma} E_{\mathbb{X} \mid \mathcal{D},\mu,\sigma} \left[\log P(\mathbb{X} \mid \mu, \sigma)\right] \tag{3.12}$$

where $\mathcal{D}$ refers to a training dataset that contains values of $\mathbb{X}_{k-1}$, $\mathbb{X}_k$, and $\mathbb{X}_{k+1}$ under normal operation as well as during FIs. Here, computation of Z is intractable because of the combinatorially large size of $\mathbb{X}_{k-1} \times \mathbb{X}_k \times \mathbb{X}_{k+1}$. However, eq. (3.9) does not require the computation of Z, as it is a common multiplicand to all values of the objective function.

Training Data. The variables in $\mathbb{X}_k$ are measured by executing the ADS in several driving scenarios in a simulator. We describe the setup of this simulator in §3.4. Simply capturing the data under normal operation is not sufficient to capture abnormalities created in the ADS state because of faults. Therefore, in addition to running driving scenarios without faults, we run the driving scenarios while injecting random faults (i.e., the baseline described in §3.2) one at a time. The FI campaign that corresponds to the training data is described in §3.5. We recreate the process of injecting a fault into a uniformly randomly selected scene 20 to 50 times for each fault. The reason for varying the number of faults is that some variables (such as ζ, b, and ϕ) exhibit all possible values during simulated runs with no injections, while others, such as stateful variables, simply do not vary naturally.

Fault Injection. The computation of $\mathbf{F}_{\text{crit}}$ (from eq. (3.1)) is done offline for every frame in every driving scene. The FI procedure executes as follows (see Fig. 3.7):

- For each driving scenario, a non-fault-injected "golden" execution of the simulation is performed. At each instant k, the variables in $\mathbb{X}_k$ are measured and stored.
- These "golden" values of $\mathbb{X}_k$ are stepped through with eq. (3.10) to build $\mathbf{F}_{\text{crit}}^{(\mathbf{k})}$ for every scene/frame, based on eq. (3.1).

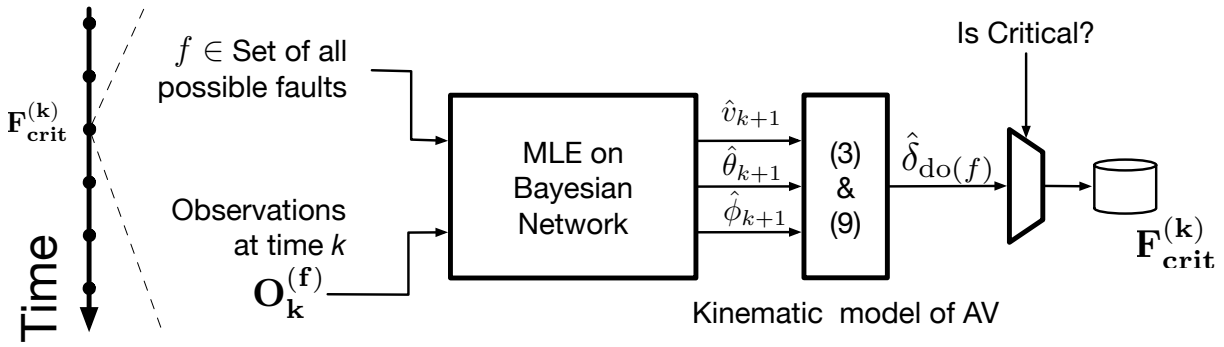


Figure 3.7: BN MLE inference is executed offline for every simulated time point to find the set of critical faults.

- An FI campaign is carried out on the simulated EV to execute faults in $\bigcup_k \mathbf{F}^{(\mathbf{k})}_{\text{crit}}$ one frame and one fault at a time.

3.4 THE ADS ARCHITECTURE & SIMULATION

3.4.1 AI Platform

An AV uses ADS technology to support and replace a human driver for the tasks of controlling the vehicle's steering, acceleration, and monitoring of the surrounding environment (e.g., other vehicles/pedestrians) [77]. The ADS architecture consists of five basic layers [15], discussed below:

Sensor Abstraction Layer (❶ in Fig. 3.8): The sensor abstraction layer is responsible for preprocessing of input data, noise filtering, gains control [134], tone-mapping [135], demosaicking [136], and extraction of regions of interest, depending on the sensor type. An ADS supports a wide range of sensors, such as Global Positioning System (GPS), Inertial measurement unit (IMU), sonar, RADAR, LiDAR, and camera sensors. Our experiments only use two cameras (fitted at the top and front of the vehicle) and one LiDAR.

Perception Layer (❷ in Fig. 3.8): The sensor abstraction layer feeds data into the perception layer, which uses computer vision techniques (including deep learning [137]) to detect static objects (e.g., lanes, traffic signs, barriers) and dynamic objects (e.g., passenger vehicles, trucks, cyclists, pedestrians) present in a driving scenario.

The object detection algorithm performs several tasks (e.g., segmentation, classification, and clustering). It uses all the sensor data separately and then merges the data using sensor fusion algorithms (e.g., extended Kalman filtering [138, 127]). The fusion

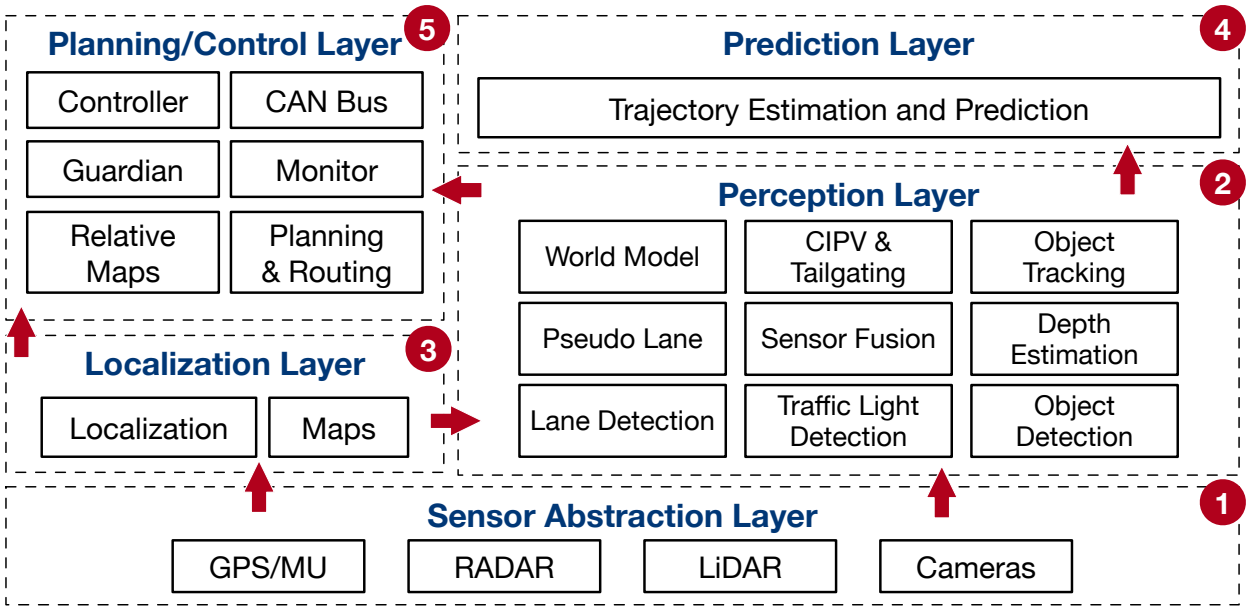


Figure 3.8: ADS architecture.

algorithm provides software-level data redundancy for object detection. Use of HD maps and the localization module enables the ADS to predetermine the location of specific static objects, such as traffic lights, further improving the confidence in obstacle-detection tasks.

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

Localization Layer (3 in Fig. 3.8): The localization module is responsible for aggregating data from various sources to locate the autonomous vehicle in the world model. Localization in the world model can be done using a GPS sensor or by using camera/LiDAR inputs. The work described in this chapter uses only camera/LiDAR along with maps to enable localization (i.e., it does not use GPS).

Prediction Layer (4 in Fig. 3.8): The prediction layer is responsible for generating trajectories for detected objects by using information from the world model (e.g., positions, headings, velocities, accelerations). As a result, it can probabilistically identify obstacles in an AV's path [139].

Planning & Control Layer (❺ in Fig. 3.8): The planning and control layer is responsible for generating navigation plans based on the origin and destination of the EV and for sending control signals (actuation, brake, steer) to the AV. The "Routing module" generates high-level navigation information based on requests. The Routing module needs to know the routing start point and routing end point, in order to compute the passage lanes and roads. The "Planning module" plans a safe and collision-free trajectory by using localization output, prediction output, and routing output. The "Control module" takes the planned trajectory as input and generates the control command to pass to the CAN Bus, which passes the information to the AV's mechanical components. The surveillance system monitors all the modules in the vehicle, including hardware. The "Monitor module" receives data from different modules and passes them on to a human-machine interface for the human driver to view to ensure that all the modules are operating normally. In the event of a module or hardware failure, the monitor triggers an alert in the "Guardian module," which then chooses an action to be taken to prevent an accident.

3.4.2 Simulation Platform

This chapter uses Unreal Engine (UE) based simulation platforms (Carla [140] and DriveSim [141]) that are capable of simulating complex urban and freeway driving scenarios by using a library of urban layouts, buildings, pedestrians, vehicles, and weather conditions (e.g., sunny, rainy, and foggy). The simulation platforms are capable of generating sensor data at regular intervals (from cameras and LiDARs) that can be fed to the ADS platform. A driving scenario consists of 500 scenes in DriveAV or 2400 scenes in Apollo in which the EV travels from a fixed starting point on the road to a fixed destination point. A scene in a driving scenario is a representation of the physical world at the simulation epoch and corresponds to a camera frame. Fig. Fig. 3.9 illustrates scenes from three freeway (DS1–DS3) and three urban (DS4–DS6) driving scenarios used in this study. DS1–3 are controlled by DriveAV in DriveSim, and DS4–6 by Apollo in Carla. In these scenarios an EV and a few UE-controlled TVs/pedestrians are placed in urban and freeway roads, driving at different velocities/accelerations and separated by some distance. The EV is expected to execute driving maneuvers in each of these settings. The scenarios represent the most common driving cases encountered by humans on a daily basis. In DS1–DS6, the Ego vehicle does not switch lanes, there is no other vehicle trailing the Ego vehicle, and the Ego vehicle is in a safe state.

3.4.3 Hardware Platform

The NVIDIA DriveAV ADS was designed for the NVIDIA AGX Pegasus platform [142],

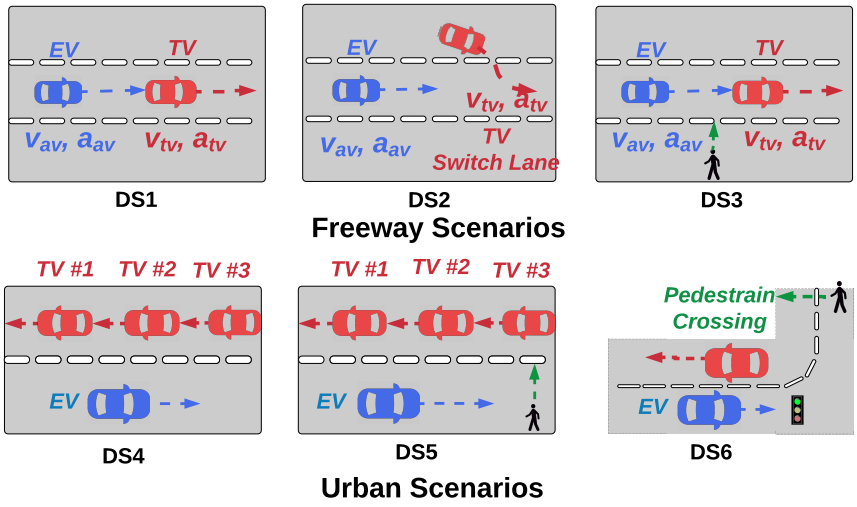


Figure 3.9: Driving scenarios supported by simulation engine.

which consists of two Xavier SoCs and two discrete GPUs, but is also supported on a development platform based on an x86 CPU and a GPU. For our experiments, we used the development platform and its utilities to facilitate the creation of the DriveFI tool. The Apollo ADS is supported on the Nuvo-6108GC [143], which consists of Intel Xeon CPUs and NVIDIA GPUs. We use Apollo on an x86 workstation with two NVIDIA Titan Xp GPUs.

3.5 DRIVEFI ARCHITECTURE

The software architecture of DriveFI is shown in Fig. 3.10. DriveFI leverages the existing tools to simulate driving scenarios and control the EV in simulation by using an AI agent (which is provided by Apollo or DriveAV). The scenario manager coordinates the simulator and AI agent to run a driving scenario and monitor the state of the software as well as the safety of the EV. DriveFI is bundled with a campaign manager that takes an XML configuration file as input to select a fault model, software or hardware module sites for FI, the number of faults, and a driving scenario. The campaign manager uses the specified configuration to (a) profile the ADS workload, (b) generate a fault plan[4], and (c) inject one or more transient faults per run into the ADS system. Based on the values in the configuration file, the campaign manager runs a specified number of golden simulations, profiles the ADS while running a driving scenario, and runs a specified number of

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

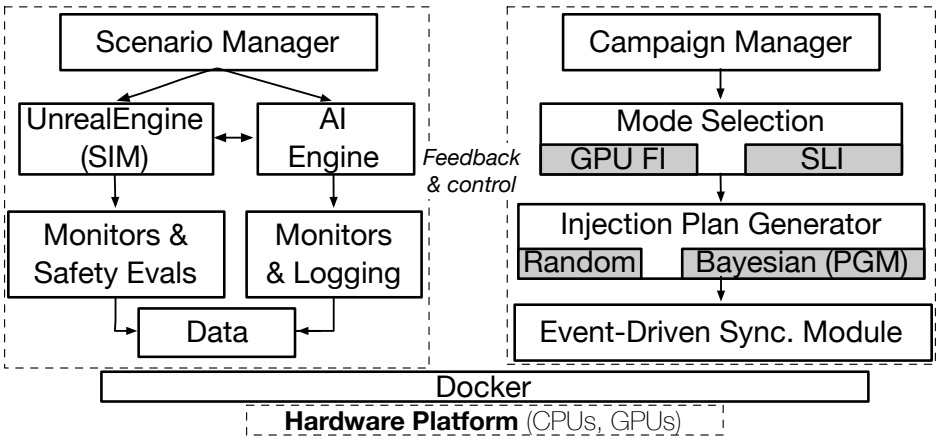


Figure 3.10: DriveFI architecture.

experiments that inject one or more faults at a time based on the generated fault plan. The "Event-driven synchronization" module helps coordinate among all the toolkits (the UE-based driving scenario simulator, monitoring agents, campaign manager, fault injectors, and AI agent).

We built DriveFI to characterize error propagation and masking (a) in computational elements, (b) in the ADS, and (c) in vehicle dynamics and traffic. Low-level circuit, micro-architectural, and RTL faults manifest as architectural-state faults in computational elements. The architectural-state faults that do not get masked manifest as errors in the internal state of the ADS modules, and the errors that do not get masked in the module propagate to the output of the module. Finally, errors that are not masked in any of the modules manifest as actuation command errors that are sent to the AV. Therefore, to mimic faults and errors, we built two fault injectors: (a) a GPU fault injector (GI; see Section §3.5.1) capable of injecting faults into the GPU architecture state to reveal the propagation of GPU faults to the ADS state, and (b) a source-level fault injector (SLI, see Section §3.5.2) capable of injecting faults to corrupt ADS software variables. Corruption of the final output (actuation values ζ, b, θ) of the ADS helps us to measure the resilience associated with vehicle dynamics and traffic. Thus, our approach aids in the measurement of fault and error masking/propagation at different levels and the corresponding impact on the safety of the AV.

DriveFI is bundled with a campaign manager that takes an XML configuration file as input to select a fault model, software or hardware module sites for FI, the number of faults, and a driving scenario. The campaign manager uses the specified configuration (a) to profile the ADS workload, (b) to generate a fault plan, and (c) to inject one or more transient faults per run into the ADS system. For this work, we developed an "event-

driven synchronization" module that coordinates among all the toolkits (the UE-based driving scenario simulator, monitoring agents, campaign manager, fault injectors, and AI agent).

3.5.1 Injecting into Computational Elements: GPU Fault Models

We consider transient faults in the functional units (e.g., arithmetic and logic units, and load store units), latches, and unprotected SRAM structures of the GPU processor. Such transient faults are modeled by injecting bit-flips (single and double) in the outputs of executing instructions. If the destination register is a general-purpose register or a condition code, one or two bits are randomly selected to be flipped. For store instructions, we flip a randomly selected bit (or bits) in the stored value. Since we inject faults directly into the live state (destination registers), our fault model does not account for various masking factors in the lower layers of the hardware stack, such as circuit-, gate-, and micro-architecture-level masking, as well as masking due to faults in architecturally untouched values. The GI employs an approach similar to that of SASSIFI [118] and includes a profiling pass and fault-injection plan generation. We do not consider faults in cache, memory, and register files, as they are protected by ECC.

Table 3.1: Examples of SLI-supported ADS module outputs.

FI Target (Output Variables)
Path Perception Module
`lane_type, lane_width`
Object Perception Module
`camera_object_distance, camera_object_class, lidar_object_distance, lidar_object_class, sensor_fused_obstacle_distance, sensor_fused_obstacle_class`
Planning & Control Module
`vehicle_state_measurements` (pos, v, a)`, obstacle_state_measurements` (pos, v, a)`, actuator_values` (ζ, b, ϕ)`, pid_measured_value, pid_output`

3.5.2 Injecting Faults into ADS Module Output Variables

The goal of SLI (Source-Level Injection) is to corrupt the internal state of the ADS by modifying ADS module output variables (hence, the input variables of another module) of the ADS components. SLI is implemented as a library that is statically linked to the ADS software; however, its use requires source-code modification and recompilation of

the ADS software. We did not observe any noticeable runtime difference between SLI-linked ADS and non-SLI ADS. In this work, we manually identified the software variables that store the outputs of ADS modules that play a critical role in inferring the actuation commands of the EV. Source-code modification is required in order to mark the output variable and invoke the corresponding module injector to get a corrupted value by using the fault model provided in the XML config file. In Table 3.1, we show some of the variables from each of the ADS modules (see Fig. Fig. 3.8) that were targeted using SLI.

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

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

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

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

M-Random. m faults are injected in a set of randomly chosen ADS software module output starting at scene k, and continues to inject faults in the ADS software module

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

Table 3.2: Fault injection experiments.

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

output until scene $K + M$. m is chosen uniformly and randomly between 10 and 100. In this case, both the ADS module and the corruption value are selected uniformly and randomly.

3.6 RESULTS

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

Because of space restrictions, without any loss of generality, we limit our discussion to DS1, in which the EV was controlled by DriveAV, and DS6, in which the EV was con-

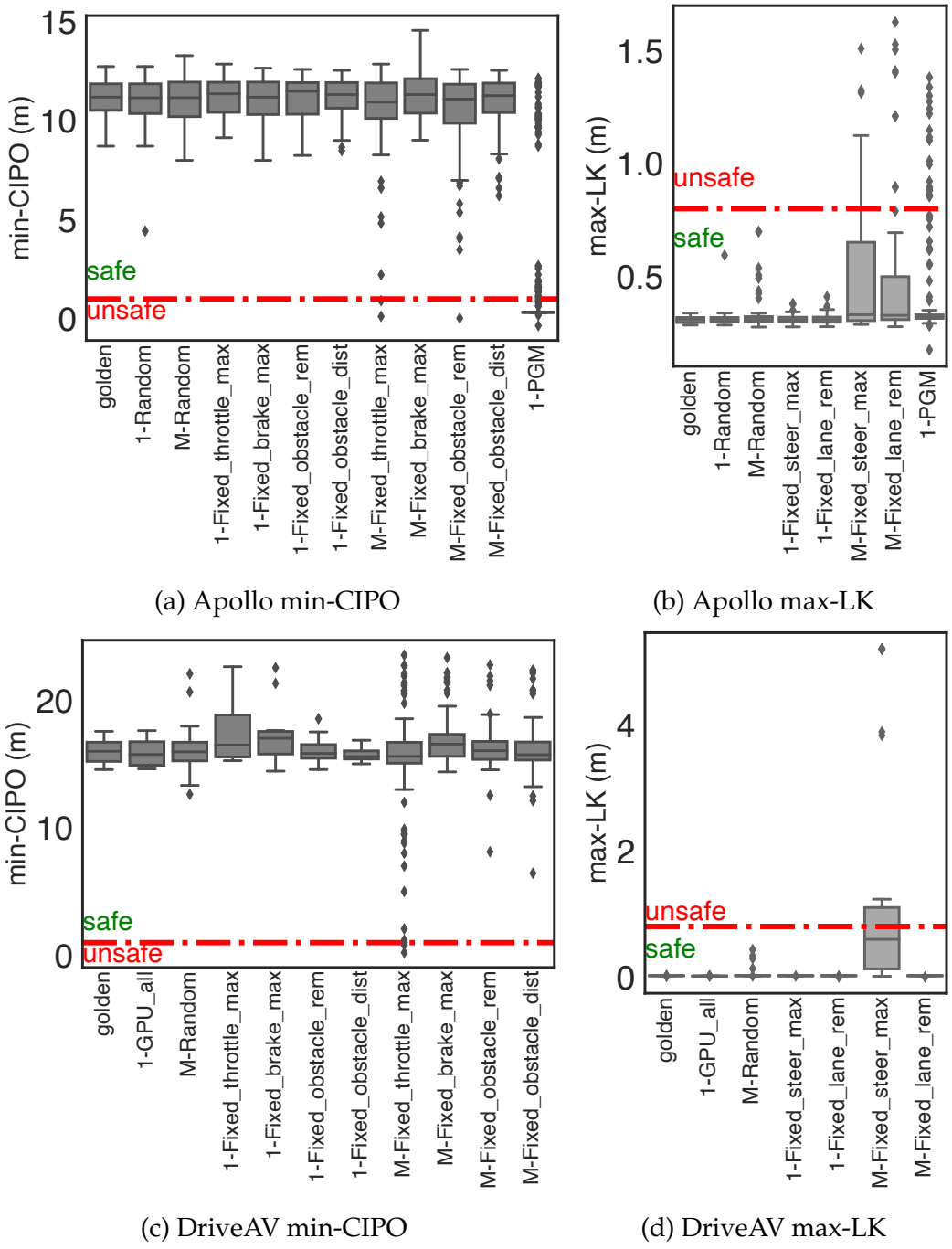
trolled by Apollo. Figs. 3.11a–3.11d show the boxplots of min-CIPO and max-LK for Apollo (DS6) and DriveAV (DS1), respectively, across all fault injection experiments and golden runs. These experiments are summarized in Table 3.2. A boxplot shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. The boxplot shows the quartiles of the dataset, while the whiskers extend to show the rest of the distribution (maximum and minimum samples), except for points that are determined to be outliers. To understand the simulation and safety characteristics of the driving scenarios, we ran 50 end-to-end simulations for each scenario without any injection. These runs are called *golden runs*. The golden runs serve as a reference against which we compare injected simulation runs in the rest of the chapter. The median min-CIPO and max-LK distances are 16 m (see "golden" in Fig. 3.11c) and 0.019 m (see "golden" in Fig. 3.11d) for DriveAV, and 11.19 m (see "golden" in Fig. 3.11a) and 0.31 m (see "golden" in Fig. 3.11b) for Apollo. None of the golden runs resulted in safety hazards.

3.6.1 GPU-level Fault Injection

We conducted 800 GPU-level FI experiments for each driving scenario (DS1, DS2, DS3) in DriveAV. The min-CIPO and max-LK of DS1 simulated in DriveAV are labelled as "1-GPU" in Fig. 3.11c and Fig. 3.11d, respectively. We conducted only 800 GPU-level FI experiments per scenario because we did not observe any safety violations during the runs, and running more experiments would have been prohibitively expensive (2.7 days per driving scenario, 800*5 minutes/FI). In FI experiments labelled "1-GPU_all", faults were chosen uniformly randomly from across all dynamic instructions in the ADS. We did not conduct any GPU FI experiments on Apollo because of a CUDA driver version mismatch between GI and Apollo. Resolving the issue would have required vendor support and fixes. From Fig. 3.11c and Fig. 3.11d, we can observe that the EV is always safe, even after FI, and that the distribution is similar to the one in the golden case.

Fault propagation and masking in GPUs. Across all GPU-FI experiments on the DS1–DS3 driving scenarios, representing a total of 2400 FI experiments, 1.9% of injected faults led to silent data corruption (i.e., caused corruption of actuation outputs which are the final outputs of the ADS module), and 0.02% led to object misclassification errors[6]. None of the object misclassification errors resulted in actuation output corruption. Our results indicate that the perception module (which is responsible for object detection and clas-

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



(a) Apollo min-CIPO

(b) Apollo max-LK

(c) DriveAV min-CIPO

(d) DriveAV max-LK

Figure 3.11: Fault/error impact characterization using FI campaigns. (a) & (b) use DS6; (c) & (d) use DS1.

sification) is more resilient than other ADS modules. The reason is that the perception software takes advantage of sensor fusion (i.e., redundancy in sensing devices can compensate for a fault of a single sensor). Across all driving scenarios, the SDCs did not result in any EV safety breach.

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

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

3.6.2 Source-level Fault Injections

We observed in the previous section that the ADS was able to compensate for injected transient faults. To further understand the ADS platform's susceptibility to faults and its robustness in the case of persistent errors, we conducted targeted FI with SLI to inject one or more faults directly into the ADS module outputs. We conducted 84 SLI-based FI campaigns for each driving scenario (scenarios 1–3 in DriveAV and 4–6 in Apollo). Of the 43 campaigns, 1 corresponded to "1-Random," 1 corresponded to "M-Random," 41 corresponded to 41 fault types under "M-Fixed," and 41 corresponded to 41 fault types under "M-Random." Labels are shown in Fig. 3.11.

Robustness of the ADS to single and multiple faults. The ADS platform was found to be robust to injection of a single fault ("1-Random" campaign). To understand the robustness to persistence of fault-generating multiple random errors, we conducted FI campaigns on driving scenarios by using "1-Random" and "M-Random" fault models. The distributions of min-CIPO and max-LK for "M-Random" were found to be statistically different from those in the golden runs for Apollo (see "M-Random" in Fig. 3.11a and Fig. 3.11b) and DriveAV (see "M-Random" in Fig. 3.11c and Fig. 3.11d). For both "1-Random" and "M-Random" campaigns, none of the injected faults led to a hazardous driving situation; however, the ADS safety was found to be more vulnerable [7] to the "M-

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

Random" fault model (especially for lane keep functionality). For example, the minimum min-CIPO observed across all injections decreased from 8.7 m to 8.0 m, and max-LK increased from 0.34 m to 0.7 m for Apollo. Similarly in DriveAV, min-CIPO increased from 15.2 m to 12.6 m, and max-LK decreased from 0.024 m to 0.43 m.

Robustness of the ADS modules to single and multiple faults. A persistent fault within the component of the ADS module continuously generates errors for the corresponding module. We tested the robustness of the ADS to a faulty module by subjecting one of the chosen module outputs to multiple faults. In these campaigns, we used "1-Fixed" and "M-Fixed" fault models. There are a total of 41 fault types for "M-Fixed" and "1-Fixed" fault types (e.g., "throttle max," "obstacle removal," and "lane removal"). We discuss the results of only select campaigns because of lack of space. The selected campaigns (shown in Fig. 3.11) included (a) actuation module output corruption (in which the brake, throttle, and steering were all changed to the "max" allowed value); (b) sensor fusion output corruption (in which the obstacle class was changed to "disappear" and the distance that could be considered in trajectory planning was changed to "max"); and (c) lane output corruption (in which the lane type was changed to "disappear"). The FI experiments that led to safety breaches appear as data points below the red line for min-CIPO and above the red line for max-LK. Clearly, none of the FI campaigns conducted under the "1-Fixed" fault model led to safety hazards, but few were observed for "M-random" FI campaigns. We rank ADS modules by their module vulnerability factor (MVF), which we calculate by finding the percent of simulations that resulted in either (a) a min-CIPO distance less than the minimum min-CIPO distance across the golden runs, or (b) a max-LK distance maximum more than the max-LK distance across golden simulation runs. Using that method, we find that the "steer angle" (MVF=46%), "lane classification" (MVF=43%), "obstacle classification" (MVF=10%), and "throttle" (MVF=7%) are most vulnerable for Apollo, whereas for DriveAV we find the same components to be vulnerable except for "lane classification" and "obstacle removal".

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

Compensation in ADS: An ADS automatically compensates for any change in EV state (i.e., θ, v, a, s) that leads to an unsafe state caused by one or more faults/errors. It does so by issuing actuation commands that bring the EV to a safe state. For example, the EV may compensate for an increased v by braking (b), a decreased v by throttling (ζ), or a change in heading angle by steering (ϕ). Fig. 3.12 shows throttle (ζ) values for golden and injected runs (in the left subfigure) and compensation achieved by braking (in the right subfigure) for an FI experiment in which ζ was corrupted in 30 consecutive frames/scenes. Compensation at time step K is calculated as the difference between the cumulative sums of "brake" values observed at time step K in the injected run and in the golden runs. The injection leads to an increase in the velocity of the vehicle, which is compensated for by braking. In the right subfigure in Fig. 3.12, we show that the compensation increases until time step $K = 232$ to undo the effects of multiple faults, and then flattens out as the brake values in the golden run and faulty run (i.e., run with fault injection) become equal. We observed similar compensation behavior for the faults injected into brake and steer values.

The ability of an ADS to compensate for injected faults depends on the number of faults and the time of injection. The outlier data point below the red line in Fig. 3.11a for "M-Fixed_throttle_max" corresponds to 30 consecutive frames/scenes injected with faults into ζ values. In this FI experiment (not shown in Fig. 3.12), the vehicle was not able to compensate for the injected faults, as the faults were injected at $K = 400$ and there was not sufficient time for the vehicle to stop, i.e., the EV reached an unsafe state at the end of the injections. In Apollo, only 20 injected faults into ζ values led to unsafe states. *Persistent errors have significant impact on the EV's state, and the ADS's ability to compensate for the impact of errors depends on the time and location of FIs.*

3.6.3 Results of Bayesian FI-based injections

In our FI campaigns thus far, hazardous driving conditions (accidents and lane violations) were created only when multiple faults had been injected into the ADS (i.e., multiple consecutive frames/scenes had been injected). However, in the real world, it is more likely that a single fault will occur, and therefore it is important to find conditions under which a single fault can lead to hazardous driving conditions. One way to approach the problem of finding all such single faults (i.e., critical faults) is to inject every single fault while running a driving scenario in a simulator. That approach, however, would be prohibitively costly and is infeasible in practice. For example, an exhaustive search to find which of the 41 fault types under the "1-Fixed" fault model will lead to safety

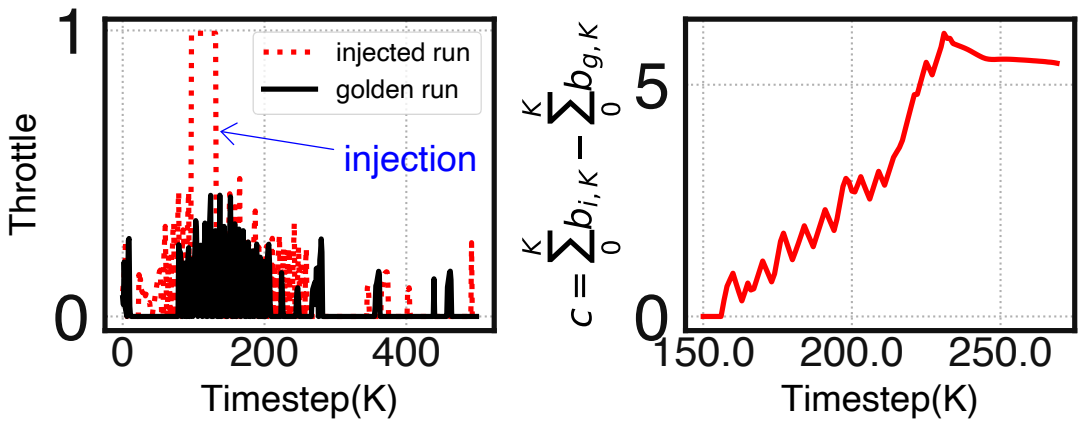


Figure 3.12: Impact of 30 continuous faults on ζ in DriveAV. Left subfigure shows ζ for a golden simulation (in black) and an injected simulation (in red). Right subfigure shows compensation c.

hazards would have taken 272 days[8] [9] in our simulation platform . Another way to find critical faults is to inject faults uniformly and randomly. However, the results from GPU hardware-level FI (see §3.6.1) and ADS software module-level FI (see §3.6.2) suggest that we need a smart FI method capable of identifying hazardous situations in driving scenarios and using them to guide FI experiments. A fault injector based on such a method would inject a fault when the ADS is most vulnerable (i.e., the fault is likely to propagate to actuators) and in such a way that the ADS cannot compensate for the fault. The Bayesian fault injector is able to find a *critical situation* that was inherently safe (i.e., $\delta > 0$) but became unsafe after injection of fault f (i.e., $\delta_{\text{do}(f)} \leq 0$). We have shown the effectiveness of Bayesian FI by injecting faults into driving scenarios DS4-DS6 controlled by Apollo.

Effectiveness of Bayesian FI. When we used Bayesian FI, 82% of injected faults resulted in hazards. (95% of the hazards were accidents involving a pedestrian, and 5% were lane violations.) Bayesian FI selects one of the 41 fault types of the "1-Fixed" fault model, and uses SLI to inject a single fault into an ADS module output variable. Recall that in the"1-Fixed' fault model, the fault location (i.e., the ADS module output variable) and corruption value are defined by the fault type. In comparison, none of the random single FIs led to safety hazards. The Bayesian FI results are marked as "1-PGM" in Fig. 3.11a and Fig. 3.11b. All data points below the red line in Fig. 3.11a correspond to collisions, and all data points above the red line in Fig. 3.11b correspond to lane violations.

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

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

The median min-CIPO distance was 0.32 m, which is significantly less than the 11.19 m median value for golden runs. Although the median max-LK value did not change for the "1-PGM" campaign compared to golden runs, 5% of the hazards were due to lane violations.

Mining critical faults and critical scenes. As discussed before, injection of all fault types under the "1-Fixed" fault model of SLI would be prohibitively expensive. Bayesian FI helped us find all critical faults $|F_{crit}|$ for every scene and mine driving scenes that are more susceptible to faults. The critical faults mined by Bayesian FI can help designers understand the weaknesses of the system and corner cases under which a fault may lead to hazards, whereas the critical scenes can be used by designers to inject random faults (using GI or SLI) only in those scenes to help them understand the architecture vulnerability factor (AVF). We believe that the mining of critical scenes by Bayesian FI will have wider applicability beyond our FIs here. Combination of results from a range of FI experiments to create a library of scenes will help manufacturers develop rules and conditions for AV testing and safe driving. Table 3.3 gives summary statistics of mined critical faults and scenes in the driving scenarios (DS4–6). A total of 561 faults were found to be critical across DS4–6. Upon inspecting the mined critical faults, we found that the top 3 most susceptible ADS module outputs for vehicle collision are the throttle value (24% of 561 critical faults), the PID controller input (18%), and the sensor-fusion obstacle class value (15% of 561 critical faults). ADS module outputs targeted by Bayesian FI for creating lane violations are the (a) lane type value (2% of 561), (b) throttle (1.4%), and (c) steer (1.4%). 56% of the fault types were never used by Bayesian FI; for example, Bayesian FI never injected into the output of camera-sensor object classification module.

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

Table 3.3: Summary of PGM-based fault injection.

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

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

to follow the lead vehicle when the lane markings are missing. However, in turns for which there is no lead vehicle to follow, such errors become critical. *It is worthwhile to note that Bayesian FI was able to mine critical faults and scenes in 4 hours, and took approximately 54 hours to simulate all the extracted faults in the simulator.*

3.7 RELATED WORK

AV research has traditionally focused on improvement ML/AI techniques. However, as models are deployed at large scale on computing platforms, the focus changes to assessment of the resilience and safety features of the compute stack that drives the AV. Assessment of the safety and resilience of AVs requires robust testing techniques that are scalable and directly applicable in real-world driving scenarios. It is not scalable or practical to base a safety argument solely on statistical measures such as a billion miles on roads, or on simulations done on platforms such as CARLA [140] or Open Pilot [16], [69, 86]. Testing the robustness of an ADS has proven to be challenging and mostly ad hoc or experience-based [112]. In particular, to test the functionality and design of the hardware and software components of an ADS, current methods rely on injection of invalid or perturbed inputs [120, 123, 8] or faults and errors [8, 75, 144] into an ADS in simulation or ADS components, and accrual of millions of miles on roads [20].

However, these methods are not scalable because (a) they lack simulated or real datasets that would represent all kinds of driving scenarios [69]; (b) it would take billions of miles of driving to add functionality or do a bug fix, in order to drive statistical measures [145]; (c) they are restricted to DNNs[146, 147, 86, 148, 75] and sensors [8, 123], even when DNNs form only a small part of the whole ecosystem; and (d) once the easy bugs have been fixed, finding rare hazardous events would be exponentially more expensive, as

faults might manifest only under specific conditions (e.g., a certain software state).

3.8 CONCLUSION

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

CHAPTER 4: AV: CRAFTING DOMAIN-GUIDED ML-DRIVEN MALWARE

Ensuring the safety of autonomous vehicles (AVs) is critical for their mass deployment and public adoption. However, security attacks that violate safety constraints and cause accidents are a significant deterrent to achieving public trust in AVs, which hinders a vendors' ability to deploy the AVs. Creating a security hazard that results in a severe safety compromise (for example, an accident) is compelling from an attacker's perspective. In this chapter, we introduce an attack model, a method to deploy the attack in the form of a smart malware, and an experimental evaluation of its impact on production-grade autonomous driving software. We find that determining the time interval during which to launch the attack is critically important for causing safety hazards (such as collision) with a high degree of success. For example, the smart malware caused $\mathbf{33}\times$ more forced emergency braking compared to random attacks, and accidents in 52.6% of the driving simulations.

4.1 INTRODUCTION

Autonomous vehicle (AV) technologies are advertised to be transformative, with a potential for bringing greater convenience, improved productivity, and safer roads [58]. Ensuring the safety of AVs is critical for their mass deployment and public adoption. However, security attacks that violate safety constraints and cause accidents are a significant deterrent to achieving public trust in AVs, which also hinders a vendors' ability to deploy the AVs. Creating a security hazard that results in a serious safety compromise (for example, an accident) is attractive from an attacker's perspective. For example, smart malware can modify sensor data at an opportune time to interfere with the inference logic of an AV's perception module. The intention is to miscalculate the trajectories of other vehicles and pedestrians, leading to unsafe driving decisions and consequences. Such malware can fool an AV into inferring that an in-path vehicle is moving out of the lane, while in reality, the vehicle is slowing down, which can lead to a serious accident.

This chapter introduce i) the foregoing attack model, ii) a method to deploy the attack in the form of a smart malware (RoboTack), and iii) an experimental evaluation of its impact on production grade autonomous driving software. Specifically, the proposed attack model answers the questions of *what*, *how*, and *when* to attack. The proposed malware has a small footprint, i.e., less than 500 lines of Python/C++ code, and 4% additional GPU utilization with negligible CPU utilization in comparison to the autonomous driving stack.

This makes it difficult to detect an attack using methods that monitor the usage of system resources. The key questions addressed by RoboTack and the main contributions of this chapter are:

What to attack? RoboTack modifies sensor data of the AV to miscalculate the trajectories of other vehicles and pedestrians.

How to attack? RoboTack minimally modifies the pixels of one of the AV's camera sensors to alter the trajectory of pedestrians and other vehicles while maintaining it for a short time interval. The change in the sensor image and perceived trajectory is small enough to be considered as noise. Moreover, RoboTack overcomes compensation from other sensors (e.g., LIDAR) and temporal models (e.g., Kalman Filters).

When to attack? RoboTack employs a shallow 3-hidden layered neural network (NN) decision model to identify the most opportune time with the intent of causing a safety hazard (e.g., collisions) with a high probability of success. In contrast, adversarial learning methods [149–151, 7] focus *only* on perception (specifically, object misdetection and misclassification). We show that without *strategically* timing the attack, the success rate is negligible.

Assessment on production software. We deploy RoboTack on Apollo [15], a production-grade AV system from Baidu, to quantify the effectiveness of the proposed safety-hijacking attack by simulating ~ 2000 runs of experiments for five of the representative driving scenarios using the LGSVL simulator [152].

The *key findings* of this chapter include:

- RoboTack is significantly more successful in creating safety hazards than random attacks (our baseline). Here random attacks correspond to miscalculating the trajectory (i.e., *trajectory hijacking*) of a randomly chosen non-AV vehicle or pedestrian, at a random time, for a random duration. This is the most general condition for comparison, although we also show results for a much more restrictive set of experiments. RoboTack caused $\mathbf{33}\times$ more forced emergency braking compared to random attacks, i.e., RoboTack caused forced emergency braking in **75.2%** of the runs (640 out of 851). In comparison, random attacks caused forced emergency braking in **2.3%** (3 out of 131 driving simulations).[1]
- Random attacks caused **0** accidents, where as RoboTack caused accidents in **52.6%** of the runs (299 out of 568).
- RoboTack had higher success rate in attacking pedestrians (**84.1%** of the runs which involved pedestrians) than vehicles (**31.7%** of the runs which involved vehicles).

[1]These numbers while seemingly different are consistent as we will show in §4.6.

- Apollo's perception system is less robust towards detecting pedestrians compared to other vehicles. RoboTack *automatically* discerns this difference, hence it needs *only* **14** consecutive camera frames involving pedestrians to cause accidents, while needing **48** consecutive camera frames involving other vehicles.

Comparing RoboTack with adversarial learning. Past work has targeted deep neural networks (DNNs) used in the perception systems of the AVs to create adversarial machine-learning-based attacks [149–151, 7] and has shown adversarial results (such as misclassifying and/or misdetecting a stop sign as a yield sign). The goal of this line of research is to create adversarial objects on the road that fool the AV's perception system. However, these attacks 1. are limited because DNNs represent only a small portion of the production autonomous driving system (ADS) [15], and 2. have low safety impact due to built-in compensation provided by temporal state-models (redundancy in time) and sensor fusion (redundancy in space) in ADS, which can mask consequences of such perturbations and preserve AV safety (as shown in this chapter, and by others [146]). To summarize adversarial learning *only* tells one *what* to attack. In contrast, as we discussed in detail in §4.3.4, RoboTack tells you *what*, *when* and *how* to attack.

4.2 BACKGROUND

4.2.1 Autonomous Driving Software

We first discuss the terminologies associated with the autonomous driving system (ADS) that is used in the remainder of the chapter. Fig. 4.1 illustrates the basic control architecture of an AV (henceforth also referred to as the *Ego vehicle*, EV). The EV consists of mechanical components (e.g., throttle, brake, and steering) and actuators (e.g., electric motors) that are controlled by an *ADS*, which represent the computational (hardware and software) components of the EV. At every instant in time, t, the ADS system takes input from sensors $\mathbf{I_t}$ (e.g., cameras, LiDAR, GPS, IMU) and infers $\mathbf{W_t}$, a model of the world, which consists of the positions and velocities of objects around the EV. Using $\mathbf{W_t}$ and destination as an input, the ADS planning, routing, and control module generates actuation commands (e.g., throttle, brake, steering angle). These commands are smoothed out using a PID controller [124] to generate final actuation values $\mathbf{A_t}$ for mechanical components of the EV. The PID controller ensures that the AV does not make any sudden changes in $\mathbf{A_t}$.

4.2.2 Perception System

Definition 4.1. *Object tracking* is defined as the process of identifying an object (e.g., vehicle, pedestrian) and estimating its state s_t at time t using a series of sensor measurements

(e.g., camera frames, LIDAR pointcloud) observed over time. The *state* of the object is represented by the coordinates and the size of a "bounding box" (*bbox*) that contains the object. This estimated state at time t is used to estimate the trajectory (i.e., the velocity, acceleration, and heading) for the object.

Definition 4.2. *Multiple object tracking* (MOT) is defined as the process of estimating the state of the world denoted by $\hat{\mathbf{S}}_t = (\hat{s}_t^1, \hat{s}_t^2, ..., \hat{s}_t^{N_t})$, where N_t represents the number of objects in the world at time t, and $\hat{s}_t^i$ is the state of the i$^{\text{th}}$ object.[2]

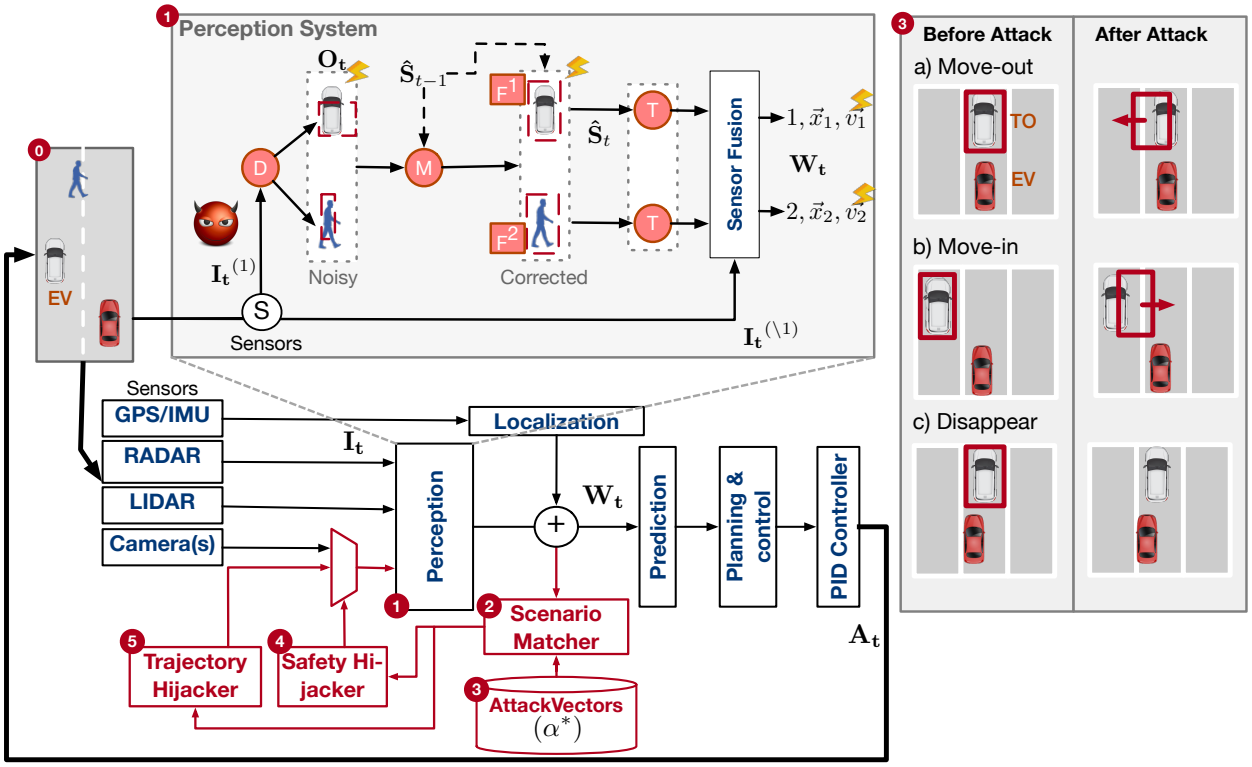


Figure 4.1: Overview of the ADS perception system and the proposed attack in RoboTack.

The MOT problem is most commonly solved by the *tracking-by-detection* paradigm [153]. An overview of this paradigm is shown in Fig. 4.1. Here, a sensor (or group of sensors) continuously collects the measurement data at every instant of time t ($\mathbf{I}_t$). These sensor inputs are sent to a corresponding DNN-based *object detector*, such as YoloNet [154] or FasterRCNN [155] (labeled as "D" in Fig. 4.1). Such an object detector estimates the object's class and its bbox at every time instant. The collection of these bbox measurements for all objects is denoted by $\mathbf{O_t} = \{o_t^1, o_t^2, ..., o_t^{M_t}\}$, where o_t^i denotes the observations for the i^{th} object at time t.

[2]In this chapter, the boldface math symbols represent tensors and regular-face symbols represent scalar values in tensors.

An *object tracker* (or tracker) tracks the changes in the position of the bboxes over successive sensor measurements. Each detected object is associated with a unique tracker, where a tracker is a Kalman filter [156] (KF) that maintains the state s^i for the i^{th} object. Each object detected at time t is either *associated* with an existing object tracker or a new object tracker, initialized for that object. Such association of a detected object with existing trackers (from time $t-1$) is formulated as a bipartite matching problem, which is solved using the Hungarian matching algorithm [157] (shown as "M" in the figure). "M" uses the overlap, in terms of IoU[3], between the detected bboxes at time t (i.e., the output of "D") and the predicted bboxes by the trackers (Kalman Filter) of the existing objects to find the matching. A KF is used to maintain the temporal state model of each object (shown as "F*" in the figure), which operates in a recursive predict-update loop: the predict step estimates the current object state according to a motion model, and the update step takes the detection results (o_t^i) as the measurement to update $\hat{s}_t^i$ state. That is, the KF uses a series of noisy measurements observed over time and produces estimates of an object state that tend to be more accurate than those based on a single measurement alone. KFs solve two real-world challenges associated with the perception system:

- Sensor inputs are captured at discrete times (i.e., $t,\ t+1, \dots$). Depending on the speed and acceleration, the object may have moved between those discrete time intervals. Motion models associated with KFs predict the new state of tracked objects from time-step $t-1$ to t.
- State-of-the-art object detectors are inherently noisy [154, 155] (i.e., bounding box estimates are approximate measurement of the ground-truth), which can corrupt the object trajectory estimation (i.e., velocity, acceleration, heading). Hence, the perception system uses KFs to compensate for the noise using Gaussian noise models [157].

Finally, a transformation operation (shown as "T" in the figure) calculates the position, velocity, and acceleration for each detected object using $\hat{\mathbf{S}}_\mathbf{t}$. These measurements are then fused with other sensor measurements (e.g., LiDAR) in the "sensor fusion" step in Fig. 4.1 to get world state $\mathbf{W}_t$.

4.2.3 Safety Model

In this chapter, we use the AV safety model provided by Jha et al. [24]. [24] defines the instantaneous safety criteria of an AV in terms of the longitudinal (i.e., direction of the

[3]Intersection over Union (IoU) is a metric to characterize the accuracy of predicted bounding boxes. It is defined as $^{\text{(area of overlap)}}/_{\text{(area of union)}}$ between the ground-truth label of the bounding box and the predicted bounding box.

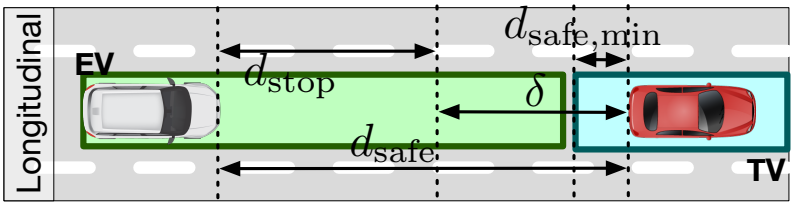


Figure 4.2: Definition of d_{stop}, d_{safe}, and δ for lateral and longitudinal movement of the car. Non-AV vehicles are labeled as *target vehicles* (TV).

vehicle's motion) and lateral (i.e., perpendicular to the direction of the vehicle's motion) Cartesian distance travelled by the AV (see Fig. 4.2). In this chapter, we only use the longitudinal definition of the safety model as our attacks are geared towards those driving scenarios. Below we reproduce the definitions of their safety model for completeness.

Definition 4.3. The *stopping distance* d_{stop} is defined as the maximum distance the vehicle will travel before coming to a complete stop, given the maximum comfortable deceleration.

Definition 4.4. The *safety envelope* d_{safe} [12, 13] of an AV is defined as the maximum distance an AV can travel without colliding with any static or dynamic object.

In this safety model, we compute d_{safe} whenever an actuation command is sent to the mechanical components of the vehicle. These ADSs generally set a minimum value of d_{safe} (i.e., $d_{\text{safe,min}}$) to ensure that a human passenger is never uncomfortable about approaching obstacles.

Definition 4.5. The *safety potential* δ is defined as $\delta = d_{\text{safe}} - d_{\text{stop}}$. An AV is defined to be in a *safe state* when $\delta > 0$.

Unlike [24] which uses $\delta \geq 0$ as the safe operation state, we choose $\delta \geq 4m$ because of a limitation in the simulation environment provided by LGSVL [152] for Apollo [15] that halts simulations for distances closer than $4m$.

4.3 ATTACK OVERVIEW & THREAT MODEL

This section describes the attacker goals, target system, and defender capabilities.

4.3.1 Attacker Goals

The ultimate goal of the attacker is to hijack object trajectories as perceived by AV to cause a safety hazard.

To be successful, the attack must:

- **Stay stealthy by masquerading attacks as noise.** To evade detection of the malicious intent, an attacker may want to hide malicious actions as events that occur naturally while driving. In our attack, we hide the data perturbation initiated by the malware/attacker as sensor noise. As we show in §4.6.1, modern object detectors naturally misclassify (i.e., identify the object class incorrectly) and misdetect (i.e., bounding boxes have zero or $< 60\%$ IoU) objects for multiple time-steps (discussed in §4.6.1). Taking advantage of this small error margin in hiding data perturbations, the attacker initiates the attack 1) at the *most opportune time* such that even if the malicious activity is detected it is too late for the defender to mitigate the attack consequences and 2) for a *short duration of time* to evade detection from the intrusion-detection systems (IDS) that monitors for spurious activities [158].
- **Situational awareness.** Hijacking the object trajectory in itself is not sufficient to cause safety violations or hazardous driving situations. An attacker must be aware of the surrounding environment to initiate the attack at the most opportune time to cause safety hazards (e.g., collision).
- **Attack automation.** An attacker can automate the process of monitoring and identifying the opportune time for an attack. This way the adversary *only* needs to install the malware instead of manually following all the steps of the attack.

4.3.2 Threat Model

In this section we discuss the target system, the existing defenses, and the attacker's capabilities.

Target system. The target is the perception system of an AV, specifically the object detection, tracking, and sensor fusion modules. To compensate for the noise in the outputs of the object detectors, the AV perception system uses temporal tracking and sensor fusion (i.e., fusion data from multiple sensors such as LIDAR, RADAR, and cameras). Temporal tracking and sensor fusion provide an inherent defense against most if not all existing adversarial attacks on detectors [146].

The critical vulnerable component of the perception system is a *Kalman Filter (KF)* (see "F" in §4.2 and Fig. 4.1). KFs generally assume that measurement noise follows a zero-mean Gaussian distribution, which is the case for the locations and sizes of bboxes produced by the object detectors (described later in §4.6.1). However, this assumption introduces a vulnerability. The KF becomes ineffective in compensating the adversarially added noise. We show in this chapter that an attacker can alter the trajectory of a perceived object by adding noise within one standard deviation of the modeled Gaussian

noise.

The challenge in attacking a KF is to maintain a small attack window (i.e., the number of contiguous time epochs for which the data is perturbed). When injecting a malicious noise pattern, the attack window must be sufficiently small (1-60 time-steps) such that the defender cannot estimate the distribution of the injected noise and hence cannot detect the attack.

What can attackers do? In this chapter we intentionally and explicitly skirt the problem of defining the threat model. Instead we focus on what an attacker could do to an AV if she has access to the ADS source code and live camera feeds.

Gain knowledge of internal ADS system. We assume the attacker has a knowledge of the internal ADS system, by analyzing the architecture and source code of open-source ADSs, e.g., Apollo [15, 159]. Attacker can also gain access to the source code through a rogue employee.

Gain access to and modify live camera feed. Recently, Argus [160] showed the steps to hijack a standalone automotive-grade Ethernet camera and spoof the camera traffic. The attack follows a "man-in-the-middle" (MITM) strategy in which an adversary gains physical access to the camera sensor data and modifies it (when certain conditions are met). The hack relied on the fact that the camera traffic is transmitted using standard (i.e., IEEE 802.1 Audio Video Bridging [161]) but simple protocols, which is not encrypted due to size of the data as well as performance and latency constraints associated with the transmission. As the camera feed is not encrypted, the attacker can reassemble packages of a video frame and decode the JFIF (JPEG File Interchange Format) payload into an image. Most importantly, since there is no hash or digital signature checks on the transmitted images, to prepare for our attack, the attacker can apply a number of filters to modify the images in-line without being noticed. The MITM attack works by using an *Ethernet tap* device to capture UDP packets in the Ethernet/RCA link between the camera and the ADS software. The Ethernet tap captures images and provides them as the input for attacker-controlled hardware with purpose-built accelerators, such as NVIDIA EGX, that are operating outside the domain of the ADS hardware/software.

Optionally compromise ADS software using secret hardware implant. To further hide malware and evade detection, an attacker can install backdoors in the hardware. Injecting malicious code in hardware-software stack has been realized in existing hardware backdoors embedded in CPUs, networking routers, and other consumer devices [162, 159]. As an AV is assembled from components supplied by hundred of vendors through a sophisticated supply chain, it is reasonable that individual components such as infotainment systems and other existing electronic component units (ECUs) can be modified to enable

secret backdoors [163, 159].

What attackers cannot do? In this work, we assume that the CAN bus transmitting the control command is protected/encrypted. Therefore we cannot launch a man-in-the-middle attack to perturb the control/actuation commands sent to the mechanical components of the EV.

Defender capabilities. In this chapter, we assume that the CAN bus transmitting the controller/actuation commands are encrypted. This is acceptable because many commercial products utilize such encryption[164]. Moreover, there are well known IDSs for monitoring CAN bus activity [158, 165]. Therefore, we do not leverage CAN bus vulnerabilities as an attack vector, instead our attack exploits vulnerabilities on the camera's Ethernet/RCA cable link.

4.3.3 Attack Vectors and Injected Attacks

We describe a taxonomy of attack vectors (shown in Fig. 4.1) that the attacker can leverage to maximize the impact, such as an emergency stop or crash. The attack vectors are:

a) Move_Out. In this attack, the attacker hijacks the target object (TO) trajectories to fool the EV into believing that the TO is moving out of the EV's lane. A close variant of this attack is fooling the EV into believing that the target object is maintaining its lane whereas in reality the target object is moving into the EV's lane. Due to this attack, EV will start to accelerate or maintain its speed, causing it to collide with the target object.

b) Move_In. In this attack, the attacker hijacks the target object (TO) trajectories to fool the EV into believing that the TO is moving into the EV's lane. Due to this attack, the EV will initiate emergency braking. The emergency braking maneuver is highly discomforting for the passengers of the EV and may lead to injuries in some cases.

c) Disappear. In this attack, the attacker hijacks the target object (TO) trajectories to fool the EV into believing that the TO has disappeared. The effects of this attack will be similar to the *move-out* attack model.

4.3.4 Attack Phases.

The attack progresses in three key phases as follows.

Phase 1. Preparing and deploying the malware. Here the attacker does the following:

1. gains access to the ADS source code,
2. defines the mapping between the attack vectors (see §4.3.3) and the world state ($\mathbf{W}_t$),

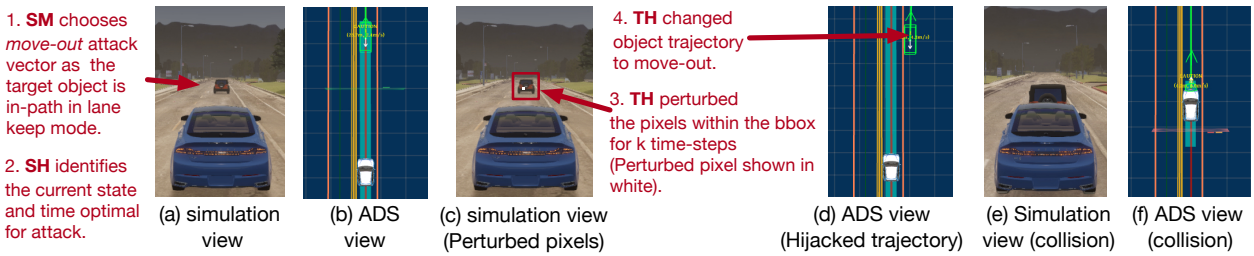


Figure 4.3: Steps followed by RoboTack to mount a successful attack, i.e., collision between the EV (blue) with the target object (red). SM: Scenario matching, SH: Safety Hijacking, TH: Trajectory Hijacking.

3. trains the 'safety hijacker' and tunes the 'trajectory hijacker' (e.g., weights of the neural network) and trajectory hijacker (e.g., learns about the maximum perturbation that can be injected to evade detection) for the given ADS,
4. gains access to the target EV camera feeds, and
5. installs RoboTack on the target EV.

Phase 2. Monitoring the environment. Once our intelligent malware is deployed, it does:

1. approximately reconstructs the world ($\mathbf{W}_t$) using the hacked camera sensor data (❶ in Fig. 4.1). For simplicity, we assume that the world state estimated using sensor fusion is not significantly different from the state determined using only one camera sensors. In our implementation we only use $\hat{\mathbf{S}}_t$ to carry the attack instead of relying on data from all sensors.
2. identifies the victim target object i (i.e., one of the other vehicle or pedestrian) for which the trajectory is hijacked. The target object is the one closest to the EV. This is done using the safety model as defined in §4.2.3 (line 9 in algorithm 4.1).
3. invokes the "scenario matcher" (SM) module (❷ in Fig. 4.1), which uses the world state ($\mathbf{W}_t$) to determine whether identified object is vulnerable to one of the the attack vectors (shown in ❸ and discussed in §4.3.3).
4. uses the "safety hijacker" (SH) (shown as ❹ in Fig. 4.1) to decide when to launch the attack (t), and for how long ($t + K$). The SH estimates the impact of the attack using a shallow 3-layered NN in terms of reduced safety potential (δ). The malware launches the attack *only* if the reduced safety potential drops below a predefined threshold (10m). We determine this threshold through simulation of driving scenarios leading to emergency braking by the EV. To evade detection, malware ensures that K does not exceed a pre-defined threshold (line 15 in algorithm 4.1). K is obtained by characterizing the continuous misdetection of an object associated with the "object detector" in

the normal (i.e., without attacks) driving scenarios executed in the simulator.

Phase 3. Triggering the attack. RoboTack:

1. uses the "trajectory hijacker" (❺ in Fig. 4.1) to corrupt the camera feed. The trajectory hijacker perturbs the camera sensor data such that i) the trajectory of the object (e.g., a school bus) is altered to match the selected attack vector (e.g., move-out) and ii) the trajectory of the object does not change significantly, thus evading detection.
2. attacks the trajectory of the victim object for next K time-steps, as decided by the safety hijacker.

4.3.5 An example of a real attack

Fig. 4.3 shows an example of a 'move-out' attack. Here we show two different views, i) simulation view, which is generated using a driving scenario simulator, and ii) ADS view, which is rendered using world-state visualizer.

RoboTack continuously monitors every camera frame using "scenario matching" (SM) to identify a target object for which the perceived trajectory by the EV can be hijacked. If SM does not identify any target object of interest, it skips the rest of the step and waits for the next camera frame. As shown in Fig. 4.3 (a) and (b), at time-step t, SM identified SUV (i.e., target vehicle) as a target object of interest, and returned 'move-out' as a matched attack-vector as the SUV is already in the Ego lane. Next, RoboTack launches 'safety hijacker' to determine the reduced safety potential of the attack, and number of time-steps the attack needs to be maintained. As it turns out the safety hijacker determined that the reduced safety potential can cause accident, and hence RoboTack launches 'trajectory hijacker' to perturb the camera sensor data as shown in Fig. 4.3 (c), and its impact on trajectory in Fig. 4.3(d). Camera sensor data is perturbed by modifying individual pixels as shown in white (within the bounding box (red square) the target object), for illustration purposes. Originally, these pixels are modified in a way that it is invisible to the human eye. Due to this attack, EV collides with the target object as shown in Fig. 4.3(e) and (f).

4.4 ALGORITHMS AND METHODOLOGY

In this section, we outline the three key steps taken by the malware: 1. in monitoring phase, selecting the candidate attack vector using the scenario matcher (§4.4.1), 2. in monitoring phase, deciding when to attack using the safety hijacker (§4.4.2), and 3. in trigger phase, perturbing camera sensor feeds using the trajectory hijacker. These steps are described in algorithm 4.1.

Algorithm 4.1 Attack procedure at each time-step.

Input: $\hat{\mathbf{S}}_{\mathbf{t-1:t-2}}$ ▷ Past object states
Input: $\mathbf{I_t^1}$ ▷ Camera feed
Global: $attack$ ▷ Flag indicating if the attack active
Global: K ▷ Number of continuous attacks
Global: i ▷ Index of the target object
Output: $\mathbf{I_t^{1'}}$ ▷ Perturbed image with adversarial patch

$\alpha \leftarrow \varnothing$
$\mathbf{O_t}, \hat{\mathbf{S}}_\mathbf{t} \leftarrow Perception(I_t)$
if $attack = False$ **then**
    $i, \delta_t \leftarrow SafetyModel(\hat{\mathbf{S}}_\mathbf{t})$ ▷ From definition 4.5
    $\vec{v}^i_{rel,t} \leftarrow calcVelcoity(\hat{s}^i_{t:t-1})$
    $\vec{a}^i_{rel,t} \leftarrow calcAcceleration(\hat{s}^i_{t:t-2})$
    $\alpha \leftarrow ScenarioMatcher(\hat{s}^i_t)$
end if
if $\alpha \neq \varnothing$ **then**
    $attack, K \leftarrow SafetyHijacker(\vec{a}^i_{rel,t}, \vec{v}^i_{rel,t}, \vec{\delta}_t, \alpha)$
end if
if $attack = True \ \wedge \ K > 0$ **then**
    $\mathbf{I_t^{1'}} \leftarrow TrajectoryHijacker(i, \mathbf{I_t^1}, o^i_t, \hat{s}^i_{t-1}, \alpha)$
    $K \leftarrow K - 1$
    if $K = 0$ **then**
        $attack \leftarrow False$
    end if
end if

4.4.1 Scenario Matcher: Selecting Target Trajectory

The goal of the scenario matcher is to check whether the closest object (referred as the target object) to the EV is vulnerable to any of the candidate attack vectors (i.e., 'Move_Out', 'Move_In', and 'Disappear'). This is a critical step for the malware to avoid launching 1) any attack if there are no objects next or in front of the EV or 2) an attack (say 'Move_Out') when the object is actually executing the would-be bogus (i.e, the selected attack vector α) driving maneuver (e.g., moving out of the Ego lane anyway). The scenario matching algorithm is intentionally designed as a rule based system (rules listed in Table 4.1), to minimize its execution time, and hence evade detection.

Note that 'Scenario Matcher' can interchangeably choose between 'Move_Out' or 'Disappear' attack vector. However in our work, we found that 'Disappear', which requires large perturbation in trajectory, is more suited for pedestrians because the attack window is small. In contrast, the attack window for vehicles is large. Therefore, in those cases RoboTack prefers to use 'Move_Out'. This is described later in detail in §4.6.

Table 4.1: Scenario Matching Map

TO trajectory	**TO in EV-lane**	**TO not in EV-lane**
Moving In	—	Move_Out/Disappear
Keep	Move_Out/Disappear	Move_In
Moving Out	Move_In	—

[1] TO: Target object

4.4.2 Safety Hijacker: Deciding When to Attack

To cause a safety violation (i.e., a crash or emergency brake), malware will optimally attack the vehicle when the attack results in $\delta \leq 4m$. Malware incorporates this insight into the safety hijacker to decide the start and stop time of the attack by executing the safety hijacker at every time-step. The safety hijacker at time-step t takes $(\vec{v}^i_{rel,t}, \vec{a}^i_{rel,t})$, δ_t, and α as inputs. It outputs the attack decision (i.e., attack or no-attack) and the number of time-steps K for which the attack must continue to be successful (line 16 in algorithm 4.1).

Let us assume that the malware has access to an oracle function f_α for a given attack vector α that predicts the future safety potential of the EV when subjected to the attack type α for k continuous time-steps,

$$\delta_{t+k} = f_\alpha(\vec{v}^i_{rel,t}, \vec{a}^i_{rel,t}, \delta_t, k). \tag{4.1}$$

Later in this section, we will describe a machine-learning formulation to approximate f_α using a neural network, and integrate it with the malware. The malware decides to attack *only* when the safety potential δ_{t+k} is less than some threshold γ. Ideally, the malware should attack when $\gamma = 4$ (i.e., δ corresponding to the crash), which causes extreme discomfort to the passengers.

In order to evade detection and masquerade the attack as noise, the installed malware should choose the 'optimal k', which we referred as K, (i.e., minimal number of consecutive camera sensor frame perturbations) using the information available at time-step t. The malware can use the oracle function $f_\alpha(.)$ to decide the optimal number of time-steps (K) for which the attack should be active. The malware decides to attack *only* if $k \leq K_{max}$, where K_{max} is the maximal number of time-steps during which a corruption of measurements cannot be detected. This is formalized in eq. (4.2).

$$K = \underset{k}{argmin}\ k \cdot (\mathbb{I}(\delta_{t+k} \leq \gamma) = 1) \tag{4.2}$$

Finally, the malware must take minimal time to arrive at the attack decision. However,

in the current formulation, calculating K can be very costly, as it needs to evaluate eq. (4.2) using f_α (which is a NN) for all $k \leq K_{max}$. We accelerate the evaluation of K by leveraging the fact that for our scenarios (§4.5.3) f_α is non-increasing with increasing k when $\vec{a}_{rel_t} \leq 0$. Hence, we can do a binary search between $k \in [0, K_{max}]$ to find K in $O(\log K_{max})$ steps.

Estimating f_α using a NN. In this work, we approximate the oracle function f_α using a feed-forward NN. We use a NN to approximate f_α to model the uncertainty in the ADS because of use of non-deterministic algorithms. Hence, the malware uses a uniquely trained NN for each attack vector. The input to the NN is a vector $[\delta_t, \vec{v}_{rel_t}, \vec{a}_{rel_t}, k]$. The model predicts δ_{t+k} after k consecutive frames given the input. Intuitively, the NN learns the behavior of the ADS (e.g., conditions for emergency braking) and kinematics under the particular attack vector that we specified, and it infers the safety potential δ_{t+k} to the targeted object from the input. We train the network using a cost function ($\mathcal{L}$) that minimizes the average $L2$ distance between the predicted δ_{t+k} and the ground-truth δ^G_{t+k} for the training dataset $\mathcal{D}_{train}$.

$$\mathcal{L} = \frac{1}{|\mathcal{D}_{train}|} \sum_{i \in \mathcal{D}_{train}} \|\delta^{G,i}_{t+k} - \delta^{i}_{t+k}\|_2^2 \tag{4.3}$$

We use a fully-connected NN with 3 hidden layers ($100, 100, 50$ neurons), ReLU activation function and dropout layers with a dropout rate of 0.1 to estimate f_α. The specific architecture of the NN was chosen to reduce the computational time for the inference with sufficient learning capacity for high accuracy. The NN predicts the safety potential after the attack within 1m and 5m for pedestrian and vehicles, respectively.

The NN was trained with a dataset $\mathcal{D}$ collected from a set of driving simulations run on Baidu's Apollo ADS. To collect training data, we run several simulation, where each simulation has a predefined δ_{inject} and a k, i.e. an attack starts as soon as the $\delta_t = \delta_{inject}$, and continues for k consecutive time-steps. Such dataset characterizes the ADS's responses to attacks. The network is trained using Adam optimizer with 60%-40% split of the dataset between the training and validation.

4.4.3 Hijacking Trajectory: Perturbing Camera Sensor Feeds

In this section, we describe the mechanism through which the malware can perturb the camera sensor feeds to successfully mount the attack (i.e., execute one of the attack vectors) once it has decided to attack the EV. The malware achieves this objective using a trajectory hijacker.

The attack vectors used in this chapter require that the malware perturb the camera

sensor data (by changing pixels) in a way that the bounding box ($\hat{s}_t^i$) estimated by the multiple-object tracker (used in the perception module) at time t moves in a given direction (left or right) at max by ω_{max}.

The objective of moving the bounding box $\hat{s}_t^i$ in a given direction (left or right) can be formulated as an optimization problem. To solve this optimization problem, we modify the model provided Jia et al.[26] to evade attack detection. We find the translation vector $\vec{\omega}_t$ at time t that maximizes the cost $\mathbf{M}$ of Hungarian matching (recall from Fig. 4.1) between the detected bounding box, o_t^i, and the existing tracker state $\hat{s}_{t-1}^i$ such that the following conditions hold:

- Threshold $\mathbf{M} \leq \lambda$ ensures that o_t^i must still be associated with its original tracker state $\hat{s}_{t-1}^i$, i.e., $\mathbf{M} \leq \lambda$. λ can be found experimentally for a given perception system and depends on Kalman parameters. This condition is relaxed when the selected attack $\alpha =$ 'Disappear'.
- $\vec{\omega}_t \in [\mu - \sigma, \mu + \sigma]$ is within the Kalman noise parameters (μ, σ) of the selected candidate object. This condition ensures that the perturbation ensures that the perturbation is within the noise.
- Threshold $IoU(o_t^i + \delta_{p,t},\ patch) \geq \tau$ ensures that the adversarial patch $patch$ should intersect with the detected bounding box, o_t^i, to restrict the search space.

$$\begin{aligned} \max_{\vec{\omega}_t} \mathbf{M}\ (o_t^i + \vec{\omega}_t,\ \hat{s}_{t-1}^i) \\ s.t.\ \mathbf{M} \leq \lambda, \\ IoU(o_t^i + \vec{\omega}_t,\ \text{patch}) \geq \gamma, \\ \vec{\omega}_t \in [\mu - \sigma, \mu + \sigma] \end{aligned} \tag{4.4}$$

Finally, the malware should stop maximizing the distance between the o_t^i and $\hat{s}_{t-1}^i$ once the total accumulated ω from start time of the attack, say $t - K'$, to current time-step t is less than Ω_{max}, i.e., $\sum_{t-K'}^{t} \vec{\omega}_t \leq \Omega_{max}$.

Once the object tracker is moved in a direction by Ω_{max}, the malware should perturb the camera sensor data to maintain the object tracker at the new location by setting $\vec{\omega}_t = 0$. Note that the trajectory hijacker maximizes the ω for *only* $K' << K$ time-steps to shift the object position at max by Ω, and maintains the position of the object for next $K - K'$ time-steps, where K is the number of time-steps for which the attack must be active from start to end. In our experiments, we find K' to be generally around 4–20 frames, whereas K (which is determined by safety hijacker) is generally 10–65 frames. Since K' is small, the chances of detection significantly decreases.

Perturbing Camera Sensor Data. Here the goal of the perturbation is to shift the po-

sition of the object detected by the object detector (e.g., YOLO). To achieve this objective, we formulate the problem of generating perturbed camera sensor data using Eq (2) given in [26]. We omit the details due lack of space.

4.5 EXPERIMENTAL SETUP

4.5.1 AI Platform

In this work, we use Apollo [15] as an AI agent for driving the AV. Apollo is built by Baidu and is openly available on GitHub [166]. However, we use LGSVL's version of Apollo 5.0 [167] as it provides features to support integration of the LGSVL simulator [152] with the Apollo. Apollo uses multiple sensors: Global Positioning System (GPS), Inertial measurement unit (IMU), RADAR, LIDAR, and camera sensors. Our experiments use only two cameras (fitted at the top and front of the vehicle) and one LiDAR.

4.5.2 Simulation Platform

In this work, we use a LGSVL simulator [152] that uses Unity [168], a gaming engine [169], for simulating driving scenarios. Note that a driving scenario is characterized by the number of actors (i.e., objects) in the world, their initial trajectories (i.e., position, velocity, acceleration, and heading), and their waypoints (i.e., their route from source to destination). In our setup, LGSVL simulates the virtual environment and posts virtual sensor data to the ADS for consumption. These sensors are LIDAR, a front-mounted main camera, a top-mounted telescope camera, continental RADAR, IMU, and GPS. The measurements for different sensors are posted at different frequencies [170]. In our experiments, RADAR is producing data at 13.5 Hz, cameras at 15Hz (of size 1920x1080), and GPS at 12.5 Hz. In simulation, LIDAR is rotation at 10 Hz and producing 360 measurements per rotation. At the time of submission of this manuscript, LGSVL does not provide integration of RADAR for Apollo. Additionally, LGSVL provides Python APIs for creating driving scenarios, which we leverage to develop the driving scenarios described next.

4.5.3 Driving Scenarios

Here we describe the driving scenarios shown in Fig. 4.4 that are used in our experiments. All our driving scenarios are generated using LGSVL on "Borregeas Avenue"

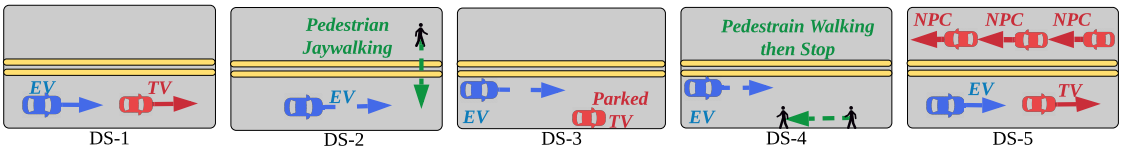


Figure 4.4: Driving scenarios. EV: Ego Vehicle, TV: Target Vehicle, NPC: Other Vehicles with no interaction with EV.

(located in Sunnyvale, California, USA), which has a speed limit of 50kph. Unless otherwise specified, in all cases EV is cruising at 45kph.

Driving scenario-1 or 'DS-1' consists of the Ego vehicle (EV) following a target vehicle (TV) in the Ego lane at a constant speed (25kph), shown in Fig. 4.4. The TV starts 60m ahead of the EV. In the golden (i.e., non-attacked) run, the EV would accelerate to 40kph and come closer to the TV at the beginning and then gradually decelerate to 25kph to match the speed of the TV. Thereafter, the EV maintains a longitudinal distance of 20m behind the TV for the rest of the scenario. We use this scenario to evaluate 'Disappear' and 'Move_Out' attack vectors on a vehicle.

Driving scenario-2 or 'DS-2' consists of a pedestrian illegally crossing the street as shown in Fig. 4.4. In the golden run, the EV brakes to avoid collision and stops more than 10m away from the pedestrian, if possible. The EV starts travelling again when the pedestrian moves off the road. We use this scenario to evaluate 'Disappear' and 'Move_Out' attack vectors on a pedestrian.

Driving scenario-3 or 'DS-3' consists of a parked target vehicle on the side of the street in the parking lane. In the golden run, the EV maintains its trajectory (lane keep). We use this scenario to evaluate 'Move_In" attack vector on a vehicle.

Driving scenario-4 or 'DS-4' consists of a pedestrian walking longitudinally towards the EV in the parking lane (next to EV lane) for 5m then standing-still for the rest of the scenario. In the golden run, EV recognizes the pedestrian, at which point it reduces its speed to 35kph. However, once it ensures that the pedestrian is safe (by evaluating its trajectory), it resumes its original speed. We use this scenario to evaluate 'Move_In' attack vector on a pedestrian.

Driving scenario-5 or 'DS-5' consists of multiple objects with random waypoints and trajectories as shown in Fig. 4.4. Throughout the scenario, the EV is set to follow a target vehicle same as 'DS-1', with multiple non-AV vehicles traveling on the other lane of the road as well as in front or behind (not shown). Apart from the target vehicle, these vehicles are traveling at random speeds starting from random positions in their lanes. We use this scenario as the baseline scenario for random attack to evaluate the effectiveness

of our attack end-to-end.

4.5.4 Hardware Platform

Production version of the Apollo ADS is supported on the Nuvo-6108GC [143], which consists of Intel Xeon CPUs and NVIDIA GPUs. In this work, we deploy Apollo on an x86 workstation with a Xeon CPU, ECC Memory, and two NVIDIA Titan Xp GPUs.

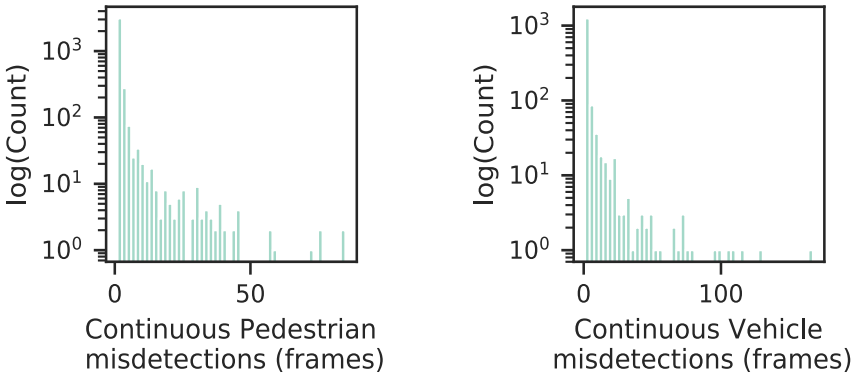
4.6 EVALUATION & DISCUSSION

4.6.1 Characterizing Perception System on Pretrained YOLOv3 in Simulation

We characterize the performance of YOLOv3 (used in the Apollo perception system) in detecting objects on the road, while the AV is driving, to measure: 1. the distribution of successive frames from an AV camera feed in which a vehicle or a pedestrian is *continuously undetected* and 2. the distribution of *error in the center positions* of the predicted bounding boxes compared to the ground-truth bounding boxes. We characterize those quantities to show that an attack mounted by RoboTack and the natural noise associated with the detector are from the same distribution. In particular, we show that the continuous misdetection caused by RoboTack is within the 99th percentile of the continuous characterized misdetection distribution of the YOLOv3 detector, see Fig. 4.5. This is important because if our attack fails, the object will reappear and be flagged by the IDS as an attack attempt. Similarly, we characterize the error in the predicted bounding box to ensure our injected noise is within the estimated Gaussian distribution parameters shown in Fig. 4.5. RoboTack changes the position at time-step t by at max $\mu - \sigma \leq \omega \leq \mu + \sigma$ of the Gaussian distribution. For this characterization, we generated a sequence of images and labels (consisting of object bounding boxes and their classes) by manually driving the vehicle on the San Francisco map for 10 minutes in simulation.

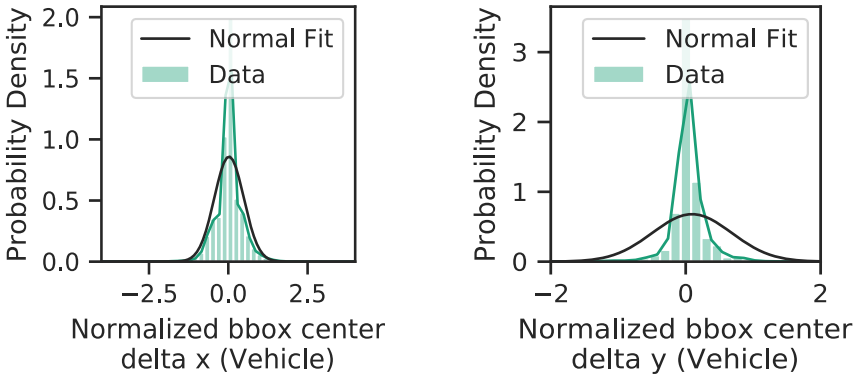
Continuous misdetections. Fig. 4.5 (a) and (b) show the distribution of the number of frames in which pedestrians and vehicles are continuously misdetected. Here we consider an object as misdetected if the IoU between the predicted and ground-truth bounding box is less than 60%. The data follows an exponential distribution.

Bounding box prediction error. To characterize the noise in the position of the bounding boxes predicted by YOLOv3, we compute the difference between the center of the predicted bounding box and the ground-truth bounding box and normalize it with re-



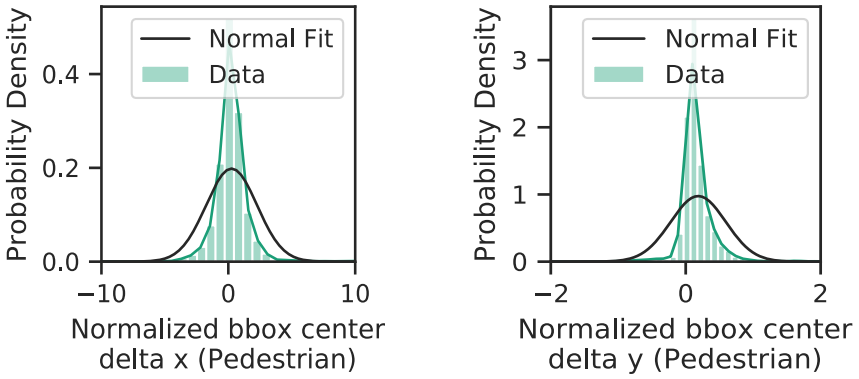
(a) Exp(loc = 1, $\lambda = 0.717$)
99th perc: 31.0

(b) Exp(loc = 1, $\lambda = 0.327$)
99th perc: 59.4



(c) Normal($\mu = 0.023$, $\sigma = 0.464$)
99th perc: 1.145

(d) Normal($\mu = 0.094$, $\sigma = 0.586$)
99th perc: 1.775



(e) Normal($\mu = 0.254$, $\sigma = 2.010$)
99th perc: 5.235

(f) Normal($\mu = 0.186$, $\sigma = 0.409$)
99th perc: 1.868

Figure 4.5: YOLOv3 object detection characterization on driving video generated using LGSVL. (a-b) show continuous misdetections with IoU=60%. (c-f) show the distribution of normalized errors in the bounding box center predictions along the x and y coordinates of the image for vehicles and pedestrians

Table 4.2: Smart malware attack summary compared with random (highlighted in bold). EB stands for emergency braking. In our experiments, the AV tried emergency braking in all runs resulting in accidents. K* means K was randomly picked between 15 and 85 for each run of the experiment.

ID	K	#runs	#EB (%)	#crashes (%)
DS-1-Disappear-TH+SH	48	101	54 (53.5%)	32 (31.7%)
DS-2-Disappear-TH+SH	14	144	136 (94.4%)	119 (82.6%)
DS-1-Move_Out-TH+SH	65	185	69 (37.3%)	32 (17.3%)
DS-2-Move_Out-TH+SH	32	138	135 (97.8%)	116 (84.1%)
DS-3-Move_In-TH+SH	48	148	140 (94.6%)	—
DS-4-Move_In-TH+SH	24	135	106 (78.5%)	—
DS-5-Baseline-Random	K*	131	3 (2.3%)	0

spect to the size of the ground-truth bounding box. Only predicted bounding boxes that overlap with the ground-truth boxes are considered. Fig. 4.5(c), (d), (e), and (f) show the distribution of normalized errors for the x (horizontal) and y (vertical) coordinates in the image of the bounding box centers for pedestrians and vehicles separately. The coordinates of the centers of the YOLOv3 predicted bounding boxes follow a Gaussian noise model.

4.6.2 Quantifying Baseline Attack Success

In the baseline attack, we perturb the camera sensor data by 1. randomly initiating the attack at time-step t of the driving scenario, 2. continuing the attack for (randomly chosen) K time-steps, 3. randomly choosing the attack vector for a simulation run, and 4. randomly choosing a vehicle or a pedestrian for which the trajectory will be changed. In other words, our baseline attack uses neither scenario matching nor the safety hijacking to mount the attack on the AV. We use 131 experimental runs of 'DS-5' in which the AV is randomly driving around the city to characterize the success of the baseline attack. Across all 131 experimental runs (see 'DS-5-Baseline-Random' Table 4.2), the AV executes an emergency braking maneuver in *only* 3 runs (2.3%) and crashes 0 times.

4.6.3 Quantifying RoboTack Attack Success

In Table 4.2, ID stands for the unique identifier for experimental campaigns, which is a concatenation of 'driving scenario id' and 'attack vector'. Here campaigns refer to a set of simulation runs executed with the same driving scenario and the attack vector. We also

append TH and SH to the ID to inform the reader that both trajectory hijacking and safety hijacking are enabled in these attacks. Other fields are K (median number of continuous perturbations), #runs (number of experimental runs), #EB (number of runs leading to AV emergency braking), and #crashes (number of runs leading to AV accidents). For each ¡driving scenario, attack vector¿ pair, we ran 150 to 200 experiments depending on the total simulation time, however some of our experimental runs were invalid due to crash of simulator or the ADS. These experiments are discarded, and only valid experiments are used for the calculations.

Across all scenarios and all attacks, we find that RoboTack is significantly more successful in creating safety hazards than are random attacks. RoboTack caused **33**$\times$ more forced emergency braking compared to random attacks, i.e., RoboTack caused forced emergency braking in in **75.2%** of the runs (640 out of 851), in comparison, random attacks caused forced emergency braking in **2.3%** (3 out of 131 driving simulations). Similarly, random attacks caused **0** accidents, where as RoboTack caused accidents in **52.6%** of the runs (299 out of 568, excluding 'Move_In' attacks). Across all our experiments, RoboTack had higher success rate in attacking pedestrians (**84.1%** of the runs which involved pedestrians) than vehicles (**31.7%** of the runs which involved vehicles).

Safety hazards with pedestrians. We observe that RoboTack is highly effective in creating safety hazards in driving scenarios 'DS-2' and 'DS-4', which involve pedestrians. Here we observe that in 'DS-2' with 'Move_Out' attacks, the EV collides with the pedestrian in 84.1% of the runs. Also, these attacks lead to EV emergency braking in 97.8% of the runs. In 'DS-2' with 'Disappear' attacks, the EV collides with the pedestrian in 82.6% of the runs and leads to emergency braking in 94.4% of the runs. Finally, in 'DS-4' with 'Move_In' attacks, we do not see any accidents with the pedestrian as there is no actual pedestrian on the EV lane; however, the 'Move_In' attacks lead to emergency braking in 78.5% of the runs. Note that emergency braking can be life-threatening and injurious to passengers of the EV, so it is a valid safety hazard. Interestingly, our malware needs to modify only 14 camera frames for 'DS-2' with 'Disappear' attacks and 24 frames for 'DS-4' with 'Move_In' attacks to achieve such a high success rate in creating safety hazards.

Safety hazards with vehicles. We observe that RoboTack is less successful in creating hazards involving vehicles ('DS-1' and 'DS-3') compared to creating hazards involving pedestrians. This is because LIDAR-based object detection fails to register pedestrians at a higher longitudinal distance while recognising vehicles at the same distance. Although the pedestrian is recognized in the camera, the sensor fusion delays the object registration in the EV world model due to disagreement between LIDAR and camera detections. For the same reason, RoboTack needs to perturb significantly more camera frames con-

tiguously in the case of vehicles than in the case of pedestrians. However, our injections are still within the bounds of the observed noise in object detectors for vehicles. Overall, 'Move_Out' attacks in 'DS-1' cause emergency braking and accidents in 37.3% and 17.3% of the runs, respectively, whereas for the same driving scenario, 'Disappear' attacks cause emergency braking and accidents in 53.5% and 31.7% of the runs, respectively. RoboTack was able to cause emergency braking in 94.6% of the runs using 'Move_In' attacks in the 'DS-3' driving scenario.

4.6.4 Quantifying Impact on Safety Potential

In this section, we characterize the impact of attacks mounted by RoboTack on the safety potential of the EV with and without the safety hijacker (SH). Our results indicate that the timing of the attack decided by the SH is critical for causing safety hazards with high probability of success. In particular, with SH the number of successful attacks, i.e., forced emergency braking and crashes, when hijacking the vehicle trajectories, increase by up to $\mathbf{5.1}\times$ and $\mathbf{7.2}\times$ respectively, when compared to attacks induced at random time using *only* trajectory hijacking. The corresponding increase, when the attack hijacks pedestrian trajectories are $\mathbf{14.8}\times$ and $\mathbf{24}\times$, respectively. Fig. 4.6 shows the boxplot of the minimum safety potential of the EV measured from start time of the attack to the end of the driving scenario. Recall that in our chapter, driving scenario experiencing a safety potential of less than 4m from start of the attack to the end of the attack is labelled as an accident. We determine the presence of forced emergency braking by directly reading the values from Apollo ADS. In this Fig. 4.6, TH stands for trajectory hijacking, and SH stands for safety hijacking. Boxplot labeled as 'TH' indicates RoboTack launches a trajectory-hijacking attack on EV without the safety hijacker, whereas 'TH+SH' indicates that RoboTack uses the safety hijacker to launch a trajectory-hijacking attack. Here, we omit the figures of the 'Move_In' attack vector due to space restrictions, as this does not reduce the δ but causes emergency braking.

DS-1-Disappear. RoboTack causes $7.2\times$ more crashes (31.7% vs 4.4%). Additionally, we observe RoboTack $4.6\times$ more emergency braking (53.5% vs 11.6%).

DS-1-Move_Out. RoboTack causes $6.2\times$ more crashes (17.3% vs 2.8%). Additionally, we observe RoboTack $5.1\times$ more emergency braking (37.3% vs 7.3%).

DS-2-Disappear. RoboTack causes $7.9\times$ more crashes (82.6% vs 10.4%). Additionally, we observe RoboTack $2.4\times$ more emergency braking (94.4% vs 39.4%).

DS-2-Move_Out. RoboTack causes $24\times$ more crashes (84.1% vs 3.5%). Additionally, we observe RoboTack $14.8\times$ more emergency braking (97.8% vs 6.6%).

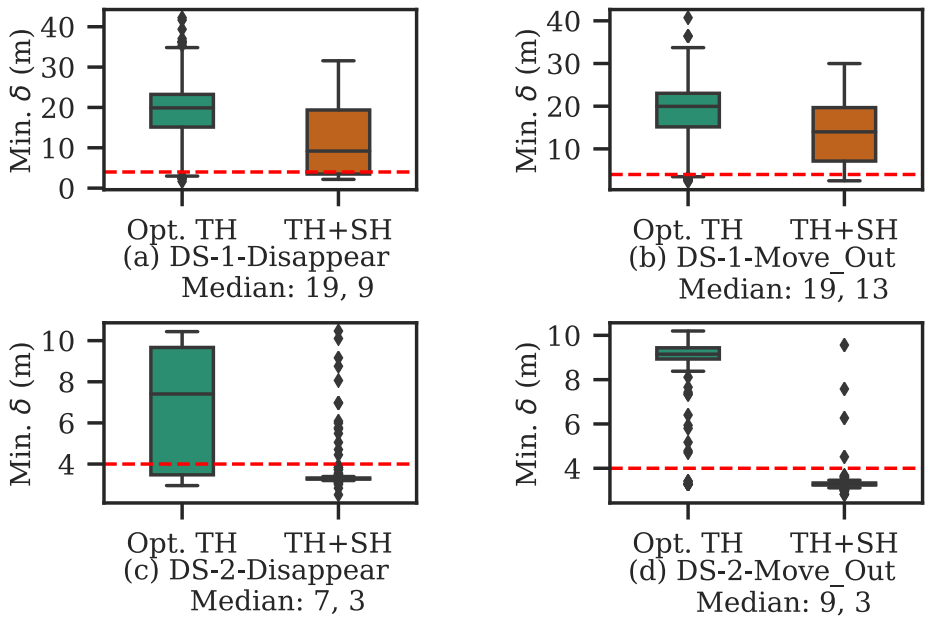


Figure 4.6: Impact of attacks. δ: Safety Potential, Opt. TH: Optimal Trajectory Hijacking, TH+SH: Trajectory Hijacking+Safety Hijacking. Red dash line indicates Min. $\delta = 4$.

DS-3-Move_In. RoboTack causes $1.9\times$ more emergency braking (94.6% vs 50%). Comparison in number of crashes does not apply as there is no real obstacle to crash into.

DS-4-Move_In. RoboTack causes $1.6\times$ more emergency braking (78.5% vs 48.1%). Comparison in number of crashes does not apply as there is no real obstacle to crash into.

Summary. In 1702 experiments (851 TH, 851 TH+SH) across all the combinations of scenarios and attack vectors, RoboTack (TH+SH) results in 640 EBs (75.2%) out of the 851 TH+SH experiments. In comparison, TH only results in 230 EBs (27.0%) out of the 851 TH experiments. RoboTack (TH+SH) results in 299 Crashes (52.6%) out of 568 TH+SH experiments excluding 'DS-3,4' with 'Move_In' attacks, while TH only results in 29 (5.1%) crashes out of the 568 TH experiments excluding 'DS-3,4' with 'Move_In' attacks.

4.6.5 Evading Attack Detection

Recall that the trajectory hijacker maximizes the ω for *only* $K' << K$ time-steps to shift the object position laterally at most by Ω, and it maintains the trajectory of the object for the next $K - K'$ time-steps, where K is the total number of time-steps for which the attack must be active from start to end. Note that RoboTack is perturbing images for all K time-steps. However for K' time-steps, RoboTack is modifying the image to change

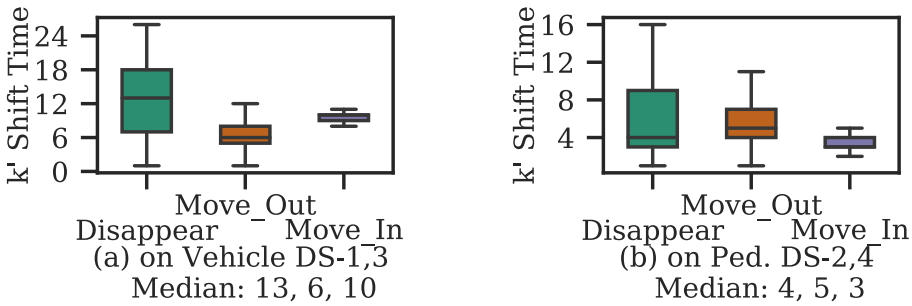


Figure 4.7: Time-steps K' required to move object in/out by Ω (a) on vehicle, (b) on pedestrian.

the trajectory, whereas for $K - K'$ time-steps it is maintaining the faked trajectory.

Fig. 4.7-(a) and Fig. 4.7-(b) characterizes K' for different scenarios and attack vectors. We observe that 'Move_Out' and 'Move_In' scenarios require small K' to change the object position to the desired location compared to the 'Disappear' attack vector. Furthermore, changing a pedestrian's location requires a lower number of time-steps compared to vehicles. When the number of time-steps K' for which the disparity between the Kalman Filter's and object detector's output is within one std. deviation of its mean, such a situation is not flagged as an attack.

4.6.6 Characterizing Safety Hijacker Performance

Here we characterize the performance of Neural Network and its impact on malware's ability to cause a safety hazard. Due to lack of space, we discuss results only for 'Move_Out'.

Fig. 4.8(b) shows a plot of the predicted value of the safety potential (using NN) and the ground-truth value of the safety potential after the attack, as obtained from our experiments. From the figure, we observe that the predicted value is close to the ground-truth value of the safety potential after the attack. On average across all driving scenarios, NN's prediction of the safety potential after the attack was within 5m and 1.5m of the ground-truth value for vehicles and pedestrians, respectively.

Fig. 4.8(a) shows a plot of successful probability (i.e., malware's ability to cause a safety hazard) on y-axis with increasing NN prediction error probability on x-axis. As expected, we find that the success probability decreases as the prediction error of the safety potential (using NN) increases. However, prediction errors are generally small.

4.7 RELATED WORK

Security attacks. AVs are notoriously easy to hack into due to i) easy physical and software access, ii) large attack vectors available to the adversary due to the complexity

and heterogeneity of the software and hardware, and iii) lack of robust methods for attack detection. Hence, the insecurity of autonomous vehicles poses one of the biggest threats to their safety and thus to their successful deployment on the roads.

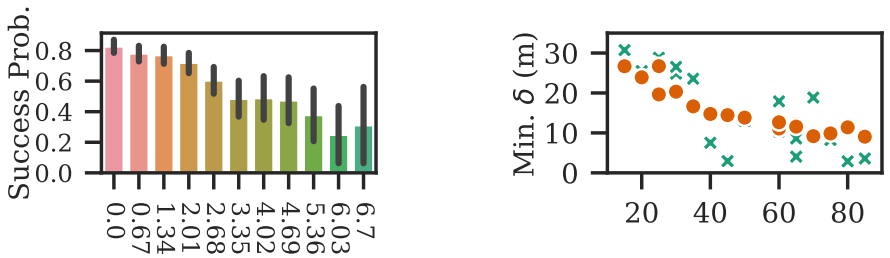
Gaining access to AVs. Hackers can gain access to the ADS by hacking existing software and hardware vulnerabilities. For example, research [162, 159] has shown that an adversary can gain access to the ADS and launch cyber attacks by hacking vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication channels [171], over-the-air software update mechanisms used by manufacturers [172], electronic component units (ECUs) [159], infotainment systems [163], and CAN buses [173]. Another possible way of hacking ADS is to use hardware-based malware, which can be implanted during the supply chain of AVs or simply by gaining physical access to the vehicle [159]. In this work, we show an attack approach that can masquerade as noise or faults and that can be implanted as a malware in either software or hardware.

Adversarial machine learning and sensor attacks in AVs. Past work has targeted the deep neural networks used in the perception systems of the AVs to create adversarial attacks [149–151, 7, 149] and has shown adversarial results (such as misclassifying and/or misdetecting a stop sign as a yield sign). A closely related work [26] targets object tracking algorithm on one camera sensor without considering i) the sensor fusion module, and ii) the control loop of the AV (i.e., they consider only statically captured video frames without running a real ADS).

The goal of research mentioned above is to create adversarial objects on the road that fool the autonomous vehicle's perception system. However, these attacks 1. are limited because deep neural networks represent only a small fraction of the overall code base of a production autonomous driving system (ADS) [15], 2. do not attempt to evade detection, and 3. have low safety impact due to built-in compensation provided by temporal state–models (redundancy in time) and sensor fusion (redundancy in space) in ADS, which can mask consequences of such perturbations and preserve AV safety (as shown in this chapter, and by others [146]).

Recently, Rubaiyat et al. [123] presented a Systems-Theoretic Process Analysis (STPA) based fault injection framework that attacks visual and RADAR modules in an open source driving agent, Openpilot [16]. However, STPA requires the developer to define hazardous situations manually in order to test the AV.

Our attack. We find that none of the above mentioned attacks is geared toward 1) evading detection by an IDS or 2) explicitly targeting the safety of the vehicles. In contrast, RoboTack is the first attack approach that has been demonstrated on production ADS software with multiple sensors (GPS, IMU, cameras, LIDAR) to achieve both ob-



(a) DS-1,2 Binned Pred. Error

(b) k, DS-1, $\delta_0 = 41m$

Figure 4.8: (a) NN binned prediction error for 'DS-1', 'DS-2' Move_Out attack. (b) 'DS-1' Move_Out attack, NN safety potential prediction, $\times$: ground-truth, $\bullet$: predicted, k: # of attacks, δ_0: starting safety potential.

jectives by digitally attacking on only one sensor (camera). RoboTack overcomes both temporal and spatial compensation by modeling the environment and safety.

4.8 CONCLUSION

In this work, we present RoboTack, a smart malware that strategically attacks autonomous vehicle perception systems to put the safety of people and property at risk. Our attack vector and malware implementation shows the ease with which smart malware can significantly impact AV safety and therefore highlights the need to develop secure AV systems.

CHAPTER 5: AV: DETECTING SAFETY-CRITICAL HARDWARE FAULTS

This chapter proposes a low-cost redundancy technique, DiverseAV, for detecting safety-critical faults in autonomous vehicles (AVs) caused by transient and permanent hardware faults. DiverseAV creates two redundant data-diverse agents by distributing the sensor data between the two agents in round-robin. The sensor data obtained between the two consecutive sequential time steps is semantically similar in terms of their worldview but significantly different at the bit-level. Thus, ensuring state and data diversity between the two agents. The data-diverse agents use the same underlying agent models (and software code) and are together responsible for driving the AV. The outputs produced by the two agents are close to each other in the fault-free case. However, in presence of a safety-critical fault, the outputs diverge significantly; thereby, enabling safety-critical fault detection. Since much of the data processing in each agent depends on the input data rate, each agent receives half the data and requires roughly half the compute resources. This allows our DiverseAV-enabled autonomous system to incur much less than 100% computational overhead compared to a fully duplicated system. Thus, in DiverseAV, we time-multiplex the two agents on the shared computational fabric.

5.1 INTRODUCTION

Autonomous vehicle (AV) technologies are advertised to be transformative, with a potential for bringing greater convenience, improved productivity, and safer roads [58]. Ensuring the safety of AVs is critical for their mass deployment and public adoption. Hardware faults caused by power-supply spikes, electrostatic discharge and external radiation strikes in the computational elements, such as CPUs, GPUs and ASICs, used in AVs pose significant threat to the safety of the vehicle. Faults may lead to a detectable uncorrectable error (DUE) that degrades system availability. Practical implementations of autonomous driving systems include a fail-back system that maintains the safety of the system in the case of a DUE. In contrast, an undetected error, i.e., a silent data corruption (SDC), may cause faulty vehicle behavior that may lead to significant safety hazards, resulting in loss of human life and serious damage to vehicles [27, 6, 24]. Future trends of increasing code complexity and shrinking feature sizes will only contribute to increasing failure rates, thereby exacerbating the problem. Thus, detecting and mitigating SDCs caused by hardware faults is important.

Current strategies for error mitigation include fault avoidance and error detection mech-

anisms such as checksums [174, 175], assertions [176–179], duplication [180–182, 181, 183–187], and data and design diversity techniques [188–190] which can be employed at the hardware or software-level. Although these strategies are largely effective, associated costs often prevent system-wide implementation. For example, hardened circuits can incur significant area and power overheads. Large SRAM arrays are often protected by ECC [191], but smaller arrays, flip-flops, and computational units are challenging to protect without significant area and power overheads. Similarly, duplication, such as at the system, module, chip, or board level, can provide high error detection coverage but also incurs significant resource and power costs.

Our Approach. We propose, DiverseAV, a novel alternative to full duplication to detect transient and permanent hardware faults that offers high error detection coverage with low performance overheads (<25%), along with corresponding power savings. Our approach requires no additional hardware and minimal modification of the AV software. DiverseAV is a lightweight, software-based redundancy technique that exploits the temporal data diversity present in the sensor data for detecting hardware faults. DiverseAV incorporates the following key ideas.

Independent and data-diverse agents. DiverseAV instantiates two independent software processes that use the same autonomous vehicle software code (referred to as AI-agent). However, unlike fully duplicated system in which agents use the exact same sensor data, DiverseAV introduces data diversity between the two agents in which the two agents use diverse data. DiverseAV leverages a key insight about the temporal semantics of the autonomous vehicle workload to introduce data diversity. In automotive workload, the sensor data obtained between the two consecutive sequential time steps is semantically similar (i.e., objects bounding box and objects do not change significantly from one time-step to another). However, the bit representation of the two dataset is significantly different: thereby, introducing data diversity between two consecutive time-steps at the instruction-level. Thus, in DiverseAV-enabled ADS, the sensor-data is distributed in round robin between the two redundant agents, thereby creating data-diverse agents. The data-diverse agents produce similar but not necessarily the same output and the divergence between the two outputs in a fault-free execution is bounded due to the similarity of the inputs in adjacent frames.

Because much of the data processing in each agent depends on the input data rate, each agent receives half the data and requires roughly half the compute resources. This allows our two-agent system to incur much less than 100% performance overhead. Thus, in DiverseAV-enabled ADS, we execute the two processes on the same computational fabric, which are together responsible for driving the AV. Since the agents are independent

processes, faults within a process propagate independently and depends on the internal process state.

Fault propagation and error detection. Since the divergence between the outputs of the two data-diverse agent is bounded in the fault-free case, DiverseAV is able to detect the error by comparing the actuator outputs (brake, throttle, and steering angle commands) of the agents. In the presence of a fault the outputs of the two agents may diverge depending on the fault type, propagation and masking in each of the individual processes (that represent software agents in execution). (i) A transient fault affects *only* one process enabling DiverseAV to detect the fault because of the independence between the agents. The fault-free agent produces fault-free outputs whereas the other agent (impacted by the fault) produces the corrupted outputs. (ii) A permanent fault that affects both processes is detectable because in the presence of a fault the two agents produce significantly different corrupted outputs as the corruption depends on the internal (private) state of and inputs to each agent, which are diverse by design.

Detecting safety-critical faults. Finally, DiverseAV aims to detect only those faults that lead to safety hazards, and also aims to detect those faults sufficiently in advance to bring the system to a safe state by using existing AV's fail-safe mechanism/support. DiverseAV uses a statistical sliding-window-based anomaly-detection algorithm to learn acceptable (bounded) divergence between the outputs of the two agents and its potential to cause a safety hazard (see §5.3).

Overall, DiverseAV is a black-box technique that offers a plug-and-play solution, requiring little to no modification to the agent itself, for achieving high coverage of transient and permanent hardware faults. It is commercially viable because it avoids software modifications to agents that are costly in terms of development and testing time. It is advantageous as it provides the state diversity needed to detect transient and permanent hardware faults at a significantly lower cost, thereby eliminating the need to fully duplicate the system.

Contributions. Our contributions include the following:

(i) We propose a novel design called DiverseAV for detecting transient and permanent errors in an AV. It has high fault detection coverage and low overhead. A key component of our design is a statistical technique for comparing the control/actuation outputs of software agents.

(ii) We have implemented the proposed design using an open-source agent [192] and an open-source simulation platform [140].

(iii) We provide an empirical characterization of temporal data diversity in onboard sensors.

(iv) Using fault injection, we have performed an experimental assessment of the functional safety of DiverseAV in fault-free operation and in the presence of faults. We also characterize the performance overhead of the proposed design.

(v) Finally, we compare the fault detection capabilities of DiverseAV-enabled ADS with a fully-duplicated system and a single-agent system.

Results. Key results include the following:

(i) *High safety.* The trajectory of the vehicle when using DiverseAV closely follows the trajectory of a single-agent-driven system. We found the maximum difference between the vehicle positions for two trajectories (i.e., those of a single agent versus the two-agent system configuration) to be <50cm, which is significantly less than the safety distance (5m) that must be maintained by the agent while driving.

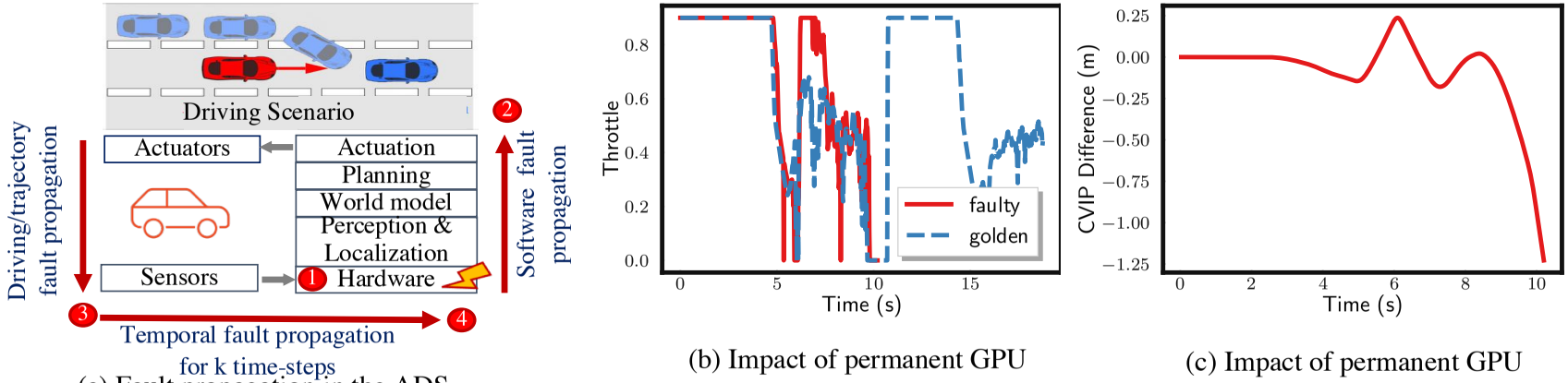
(ii) *Highly accurate.* DiverseAV detected safety-critical errors caused by the transient and permanent faults injected into the computational elements with a precision of 0.87 and a recall (which is equivalent of the detection coverage) of 0.87. DiverseAV outperforms both fully-duplicated system and single agent system (which uses temporal outlier detection techniques) in terms of accuracy. Across all our driving scenarios, neither DiverseAV nor a fully-duplicated system raised an alarm (i.e., detected an error) for fault-free experimental runs of a driving scenario. We assume availability of fail-back system that can be invoked on error to brings the vehicle to a safe state.

(iii) *Low performance overhead.* Compared to a fully duplicated system, which will incur 100% overhead, DiverseAV incurs <25% overhead.

5.2 BACKGROUND

5.2.1 Autonomous Driving Systems

Autonomous driving systems (ADS) are feedback-based control systems. Examples include self-driving cars, drones, and unmanned aerial vehicles. Fig. 5.1(a) shows the architecture of a typical ADS. An ADS continuously uses measurements from the "sensors" to infer the state of the world ("world model"), plans its trajectory ("planning"), and makes "actuation" decisions to drive the vehicle towards a set goal, all while ensuring the comfort, safety, and integrity of the passenger/vehicle and its surroundings. The control loop can be implemented as an end-to-end deep neural network (DNN) agent with a proportional–integral–derivative (PID) controller (e.g., Dave2 [193]) or as an ensemble of



(a) Fault propagation in the ADS

(b) Impact of permanent GPU fault on actuation outputs

(c) Impact of permanent GPU fault on safety

Figure 5.1: Depiction of fault propagation in autonomous driving system controlling the AI-driven vehicle. Golden represents the throttle actuation output trace corresponding to fault-free executing of the driving scenario. Faulty represents the trace corresponding to the execution of the driving scenario with GPU permanent fault injection enabled.

models (EM) agent (e.g., Baidu's Apollo [166]) in which each model is responsible for individual sub-tasks (such as perception, planning, and control). The agent must be able to execute the control loop at a very high frequency ($\sim 30 - 100$ Hz) to dynamically infer changes in the environment and react to those changes in real time. The algorithms used by the agent are computationally expensive, thereby requiring the use of a heterogeneous computational fabric consisting of CPUs, GPUs, and ASICs/FPGAs [182, 194].

5.2.2 Fault Models

ADSes can experience a range of hardware faults in the computational elements due to power-supply spikes, electrostatic discharge, external radiation strikes, and circuit degradation among others. In this chapter, we *only* consider *transient and permanent hardware fault models*, emulated via instruction-level bit-flip models, in the computational fabrics used by the ADS (such as CPUs and GPUs). We do not consider sensor fault models (caused by sensor failures or poor weather conditions) or machine-learning inference failures (caused by out-of-distribution data).

In the transient fault model, we assume that a fault corrupts the destination register of *only* one dynamic instruction[1]. In contrast, in the permanent fault model, we assume that a fault corrupts the destination register of a selected opcode for *all* dynamic instances of that opcode. The destination register is corrupted by XOR-ing the original contents of the destination register with a selected mask. In this work, we aim to detect faults (transient or permanent) that lead to safety-hazard, and do not aim to identify the fault type (i.e., differentiate between the transient or permanent fault) at runtime.

5.2.3 Impact of Faults on Safety

Hardware faults can alter the actuation decision outputs, thereby impacting the safety of the vehicle. A typical hardware fault propagation path is shown in Fig. 5.1(a). Hardware faults may corrupt the output of the hardware instruction (e.g., output of the add instruction), which in turn can corrupt the output of the software module (e.g., perception outputs). The corrupted values are then consumed by other software modules; which may ultimately taint the actuation outputs. The fault propagation in the software may also corrupt the internal state of the software (until the next reset/restart), which may result in subsequent corruption of actuation outputs in the future time-steps. The corrupted

[1]Dynamic instances of an opcode are the actual instructions of that opcode that are fetched and executed by the processor.

actuation outputs for one or more time-steps may change vehicle kinematics sufficiently enough to cause an accident.

Not all faults are hazardous to the system. Faults may result in a silent data corruption (SDC), hang, or crash. Hangs and crashes are detected by the system via exceptions and heartbeats, whereas SDCs may potentially propagate to cause safety violations. Detecting SDC-causing faults is challenging in a feedback-based control system as the errors accumulate over time (e.g., due to the PID controller). Full duplication of software and hardware ensures robust detection of both transient and permanent faults; however, they result in high resource and power overheads [182].

Fig. 5.1(b) and Fig. 5.1(c) respectively show the impact of a GPU permanent fault on the "actuation outputs" and the "safety" of the vehicle using an AI-agent (discussed in §5.4) for the driving scenario shown in Fig. 5.1(a). Fig. 5.1(b) shows that the throttle actuation outputs of the faulty run (depicted as a solid red line) is significantly different in the presence of a permanent fault from what it is in the golden (non-faulty) run (depicted as a dashed blue line). The change impacts the vehicle dynamics (velocity and acceleration), and that, in turn, reduces the safety distance between the vehicles. In this chapter, we characterize the safety distance of the AI-controlled vehicle using the closest-vehicle in-path (CVIP) distance. Fig. 5.1(c) shows the difference in CVIP distance between the golden run and the faulty run. The fault in this case leads to decrease in CVIP distance. Hence, it is of the utmost importance that we detect such faults at runtime and far enough in advance to preemptively mitigate the adverse effects of the failures (e.g., trigger fail-back system).

5.3 METHODOLOGY AND APPROACH

This section presents the DiverseAV design requirements, principles, and overall design.

5.3.1 Design Requirements

The DiverseAV design allows us to address the following design requirements.

Detection of transient and permanent faults. DiverseAV must detect transient and permanent faults that are safety-critical with high probability sufficiently far in advance. Not all faults are safety-critical (i.e., impact the software state and autonomous vehicle safety), so detecting all faults, including the faults that are masked by hardware or soft-

ware, may reduce overall system availability.

Model parameters of the error detector must be driving scenario-independent. The error detector in DiverseAV must be able to detect errors for all possible driving scenarios, and should not be limited to only those driving scenarios that were used to train the error detector model parameters.

Workload independence of the autonomous system. DiverseAV must: (i) be independent of the workload (i.e., AI agents), (ii) apply to a large class of autonomous vehicles (such as cars and trucks), and (iii) handle frequent incremental updates to software/hardware.

Plug and play design. DiverseAV must implement a plug-and-play design, thereby reducing the engineering and deployment effort.

Low cost. DiverseAV must achieve all the above properties with minimal computational and area cost overhead. For example, complete duplication of a system would incur 100% computational and area cost overhead, making the technology too costly for the end consumers.

5.3.2 Design Principles

In this work, we propose and describe our implementation of DiverseAV, which exploits the principle of temporal data diversity and redundancy. DiverseAV is an innovative redundant design for autonomous vehicles, which uses two independent software agents that are dynamic instances of the same underlying agent models (software) and are *time-multiplexed* on the shared computational fabric to actuate the vehicle. Time-multiplexing allows the following:

Semantic consistency. Each agent consumes semantically similar data. The sensing data used by the time-multiplexed redundant agents are semantically similar because the sensing frequency is typically very high, ranging from 30 Hz to 100 Hz among different sensors, and the world view (world semantics) does not change significantly between subsequent sensing time-steps.

Temporal data diversity. Temporal data diversity is enforced between the agents at the bit level (bit-level diversity). Sensor data obtained at consecutive time-steps are semantically similar but differ significantly at the bit-level. For example, a vehicle in front continues to exist at time t and $t+1$, however, the pixel values (and hence the data bits representing the car may change significantly) between the subsequent time-steps; thereby, enforcing temporal data diversity.

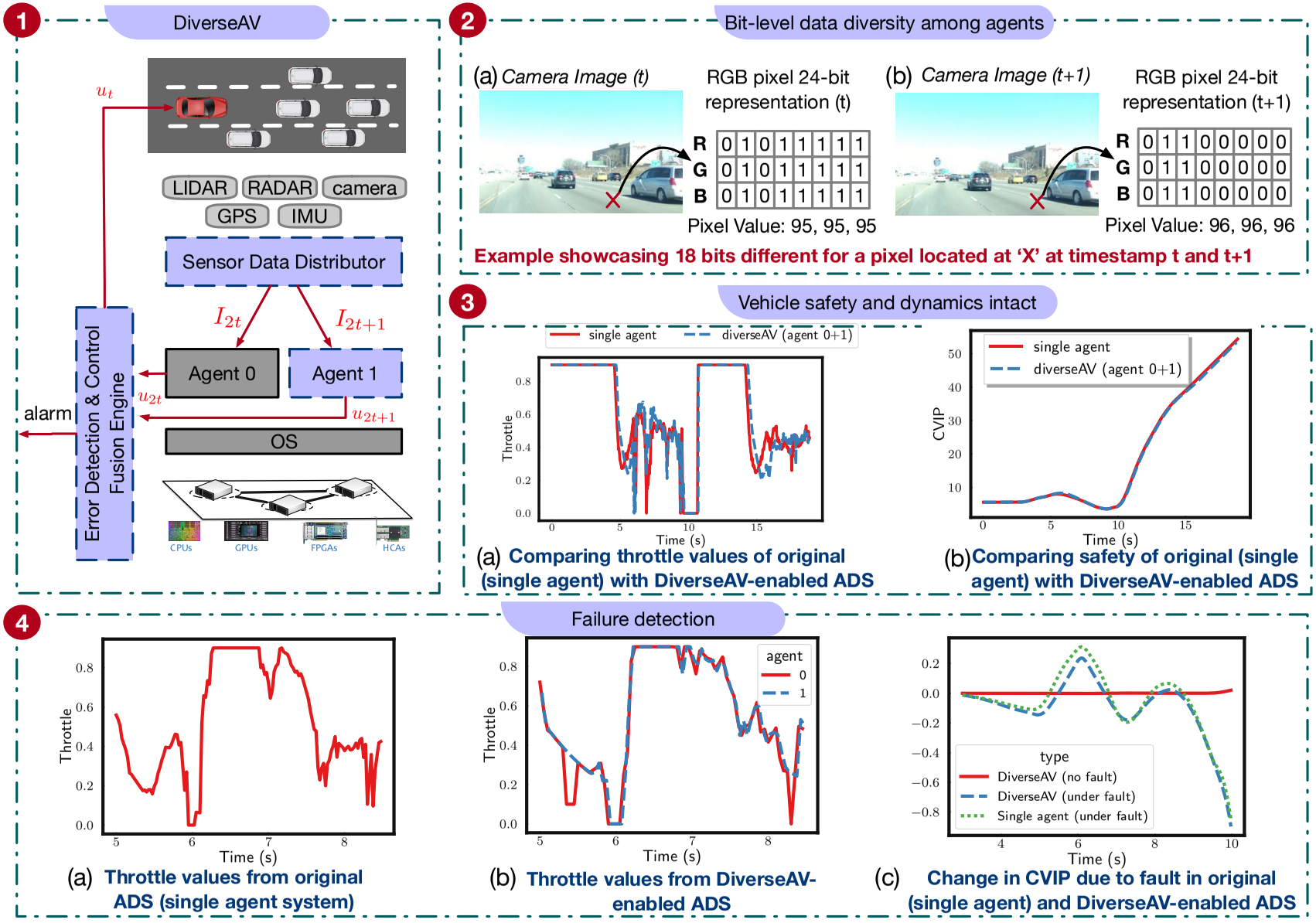


Figure 5.2: DiverseAV approach overview.

Error detection via time-multiplexing. Time-multiplexing enables detection of hardware errors that propagate from hardware to the software state and subsequently impact the AV dynamics and safety. Time-multiplexing of the sensor data between the DiverseAV-enabled agents detect a wide range of silent data corruptions, since faults either impact a single agent or impact individual agents differently. A fault manifestation in each agent may be different because each agent consumes diverse (though semantically similar) data inputs and maintains its own private state.

Model formulation.

Using above mentioned design principles, here we present the mathematical abstraction of our system. ADS can be abstracted using eq. (5.1). u_t is the actuation output (expressed as f) at time t using sensor data input I_t on a processing element (expressed as h).

$$u_t = h(f^0, I_t) \tag{5.1}$$

Since ADSes are fitted with PID-based low-level controllers, the average difference between adjacent actuation values over a sliding window are small and bounded, i.e., $\sum_{t=k}^{t=k+sw} ||u_{t+1} - u_t|| / sw \leq \delta$ as seen in Fig. 5.1(b). It is plausible to develop a monitor to find anomalies in the timeseries data measuring δ to detect errors but such a monitor is noisy leading to high false positive rates as discussed in §5.6.3.

In comparison, DiverseAV-enabled ADS can be modeled by eq. (5.2), in which the data is distributed to the agents 0 and 1, each of which execute function f, in round-robin fashion using 'sensor data distributor'. Here, $\mathbb{1}_{2k}$ and $\mathbb{1}_{2k+1}$ are indicator functions that are indicative of even and odd time steps.

$$u_t = \mathbb{1}_{2k} h(f^0, I_{2k}) + \mathbb{1}_{2k+1} h(f^1, I_{2k+1}) \tag{5.2}$$

It is easy to see that eq. (5.1) and eq. (5.2) are equivalent iff the composite function $h(f)$ is stateless. However, in practice, $h(f)$ is not stateless. However, eq. (5.2) approximates eq. (5.1) when the operating frequency of the ADS tends to infinity. This is because semantically I_t and I_{t+1} are similar even though I_t and I_{t+1} are not similar at the bit-representation level, i.e., $||w(I_{t+1}) - w(I_t)|| \rightarrow \epsilon$, where w is a function that extracts the semantic meaning from the image (e.g., bounding box of the objects or position of the object in the world), and ϵ is bounded and small. This is a fair assumption because practical implementation of ADSes operate at high frequencies (30-60Hz). However, this assumption is violated when the hardware is faulty. Under a faulty hardware (expressed as h^τ),

DiverseAV-enabled ADS can be represented by eq. (5.3).

$$u_t^\tau = \mathbb{1}_{2k} h^\tau(f^0, I_{2k}) + \mathbb{1}_{2k+1} h^\tau(f^1, I_{2k+1}) \tag{5.3}$$

Previous research [190, 195] as well as our own empirical demonstration of DiverseAV have shown that $h^\tau(f^0)$ produces significantly different outputs even when using semantically similar inputs when the input data is diverse. In our case, the input data is diverse at the bit-level which is quantified in §5.5.1. Because of this diversity, the average error between adjacent actuation outputs produced by the two agents over a sliding window is neither small nor bounded, i.e., $\sum_{t=k}^{t=k+sw} ||u_{t+1}^\tau - u_t^\tau|| / sw > \delta$; thereby, enabling us to detect the error using a statistics-driven 'Error Detection' engine.

In §5.5, we empirically demonstrate that the design decisions taken in DiverseAV does not impact safety.

5.3.3 Design Overview & Implementation

Fig. 5.2 shows the overall design of DiverseAV (❶). The modifications to the original ADS system are highlighted in boxes with dashed blue outlines. To enable time-multiplexing between the agents, we introduce a "sensor data distributor" and an "error detection and control fusion engine."

Sensor data distributor takes the sensor data as inputs ($\mathbf{I}_t$) and round-robins the input data among the two agents, thereby reducing the sensing frequency for each agent by 50%. For example, it splits the input data $\mathbf{I}_t$ such that agent 0 receives the input data $\mathbf{I}_{2t}$ and agent 1 receives the input data $\mathbf{I}_{2t+1}$, where $t \in \mathbf{N}$. Such a data distribution strategy has several advantages; it ensures that each agent uses semantically similar sensor data to compute the actuation decision while providing significant data diversity at the bit level for the two agents. As can be seen from Fig. 5.2 (❷), the subsequent camera frames captured at times $2t$ and $2t+1$ are semantically very similar; however, when they are compared at the bit level, their data are significantly different. For example, when the 24-bit RGB color value (8-bit per color) for a given pixel at location X changes from 95 (for each color at time t) to 96 (at time $t+1$), the data at the bit-level changes by 18 bits. We evaluate this temporal data diversity in detail in §5.5.1 and show that on average, there are 8 bits of difference per pixel between successive camera frames. Thus, the sensor data distributor provides the much-needed data diversity to enable error detection. However, it also introduces several complications, such as ones related to synchronization and selection of the actuation decisions produced by each agent.

Control fusion engine is responsible for synchronization of actuation decisions between the two agents. Recall from §5.2 that ADS fuses the sensor data spatially and temporally to produce actuation decisions and drive the vehicle in the real world. Depending on the ADS design, sensing and actuation can be (i) a lockstep process (i.e., an actuation decision is produced only after all inputs have been received, leading to the same sensing and actuating frequency as the original single agent system), as in the case of the Sensorimotor agent (described later in §5.4); or (ii) an asynchronous process, as in the case of Baidu's Apollo agent [166]. The Apollo agent uses an array of sensors, each operating at a different frequency (30 Hz camera, 77 Hz radar, 100 Hz GPS and IMU (Inertial Measurement Unit), and 10 Hz LIDAR) to create an internal model of the real world and continuously update the internal world model. The planning and actuation model asynchronously uses the internal world model to produce the actuation decision at 100 Hz. Implementation of DiverseAV for the above-mentioned lockstep design is straightforward: DiverseAV can use the actuation decision of the agent that received the sensor data. However, implementing DiverseAV for an asynchronous design can be challenging: with two agents, DiverseAV doubles the number of actuation decisions produced by the ADS. Furthermore, enforcing an ordering of the actuation decisions across the agents is not trivial. Therefore, for an asynchronous system, DiverseAV can either (i) use an actuation decision from only one of the agents and use the actuation decision of the other agent solely for the purposes of error detection, or (ii) use the actuation decisions of both agents by averaging the actuation decisions produced by the replicas.

Fig. 5.2(❸) shows the vehicle "throttle" actuation command value and CVIP distance ("closest-vehicle-in-path," described in §5.2) for the original system when it is using a single agent and DiverseAV-enabled ADS for the lead-slowdown driving scenario in which the lead vehicle is slowing down. Although the actuation decisions produced by DiverseAV-enabled ADS diverge from those of the original ADS by a small amount, the CVIP distance shows negligible divergence. These results are described in §5.5.2 in more detail.

Error detection engine. In a redundant system in which agents are consuming the same input data, the outputs can be compared directly using algebraic subtraction, and an alarm is raised if the subtraction yields a nonzero value. However, such systems are hard to design and implement, especially for an AI-driven system that is highly non-deterministic. Designing a redundant system that can support such subtraction-based error detection requires implementation of lockstep redundancy in both hardware and software, and that would make the system prohibitively costly. In contrast, our time-multiplexed redundant design leverages data diversity to detect a broader class of faults

and bugs with only marginal increase in computational and resource requirements. However, it also increases the difficulty in detecting safety-critical faults (i.e., faults that lead to accidents or significant divergence in vehicle kinematics) because the inputs, outputs, and internal software state is not a bit-by-bit match. Thus, the challenge is to design a robust error detector that provides high detection accuracy and lead detection time. The *detection accuracy* is measured in terms of precision ($^{\text{\#True Positives}}/_{\text{\#True Positives + \#False Positives}}$) and recall ($^{\text{\#True Positives}}/_{\text{\#Positives}}$). An error detector with higher precision and recall produces a lower number of false positives (false alarms) and misdetections. The *lead detection time* is the difference between the alarm generation time and the collision time. An error detector with a higher lead detection time allows the ADS to switch over to fail-safe mode earlier.

The outputs from the two agents may differ due to the inherent input diversity of a DiverseAV-enabled ADS and hence, we use statistical techniques for error detection, while ensuring high precision and recall. Fig. 5.2(❹) depicts the impact of a permanent GPU fault on the original ADS and DiverseAV-enabled ADS for the lead-slowdown driving scenario. One can observe that the throttle values are different in the faulty run (Fig. 5.2(❹)(a)) and a non-faulty run (shown in Fig. 5.2(❸)(a)). Since the impact of the fault is smoothed by the PID controller, there are no visible anomalies in the throttle values for the original single-agent system (Fig. 5.2(❹)(a)). However, one can see visible divergence between the outputs of the two agents in the DiverseAV-enabled ADS (Fig. 5.2(❹)(b)).

Training error detection engine. In this work, we use a sliding-window-based error detection algorithm to learn the maximum divergence between the actuation outputs of the two agents, and use that divergence as a threshold to detect errors at run time. We ensure that DiverseAV is not tuned to any specific scenarios or faults by training the error detection engine (i) using the *long training scenarios*, described in §5.4, which is significantly different from our evaluation scenarios, and (ii) by executing these scenarios under fault-free conditions.

At runtime, the DiverseAV uses the learned divergence parameters to detect an error. Upon detection of an error, an alarm is raised, and DiverseAV triggers a fail-back system with sufficient capabilities to handle the driving situation, e.g., safely park the vehicle. The sliding-window-based error detector used in DiverseAV uses the following parameters:

(i) $\theta_{throttle}(s), \theta_{brake}(s)$, and $\theta_{steer}(s)$: DiverseAV raises an alarm if the difference between the actuation command values of the two agents exceeds a certain threshold at a given vehicle state s (given by tuple $\langle v, a, \omega, \alpha \rangle$, where v is speed, a is acceleration, ω is angular velocity, and α is angular acceleration). We use $\langle v, a \rangle$ to represent the state for

$\theta_{throttle}(s), \theta_{brake}(s)$, since the throttle and brake depend on linear speed and acceleration. Similarly, we use $\langle \omega, \alpha \rangle$ to represent the state for $\theta_{steer}(s)$.
We discretize each of the variables (i.e., $\langle v, a, \omega, \alpha \rangle$) of the vehicle state s into small intervals and learn the thresholds for each of these intervals. The thresholds are learned by calculating the maximum difference between the actuation command values for the two agents across all executions of reference driving scenarios for a given vehicle state (s). The thresholds learned are stored in a lookup table (LUT), which is used at runtime for detection.

(ii) rw: The two agents in DiverseAV naturally produce slightly different actuation command values because they are consuming diverse data, and the divergence is highest when the planning decision changes between two time-steps (e.g., from slowing down to accelerating). However, such high divergence in actuation is transient. Therefore, to avoid identifying occasional blips as errors, we use a rolling window (rw–rolling window size) to smooth out the difference in actuation command values produced by the two agents. The rw parameter may impact the lead detection time. We vary the rolling window from 3 all the way to 40 recently received sensor data, as 40 Hz is the sensor frequency of our simulator, and pick the parameters for which the F1-score (harmonic mean of precision and recall) is maximum.

5.4 EXPERIMENTAL SETUP

This section describes the autonomous agent, simulation platform, driving scenarios, data collection methods, and fault injection methods used in our experiments. Hereafter, we refer to the agent-controlled vehicle as "the vehicle" and the other vehicle in a scenario as "NPC vehicle".

5.4.1 Autonomous Agent

This work uses the state-of-the-art convolutional neural network (CNN)-based end-to-end autonomous agent proposed and pretrained by Chen et al. [192], referred to as the *Sensorimotor agent*. The main components of the agent are the High-level Route Planner, CNN, Waypoints Tracker, and Control Unit. *High-level Route Planner* is responsible for finding the next "destination-to-go" navigation direction. *Convolutional Neural Network (CNN)* is a vision-based local planner, and is the core of the Sensorimotor agent. The CNN predicts the path that the vehicle should follow by outputting four local-waypoints

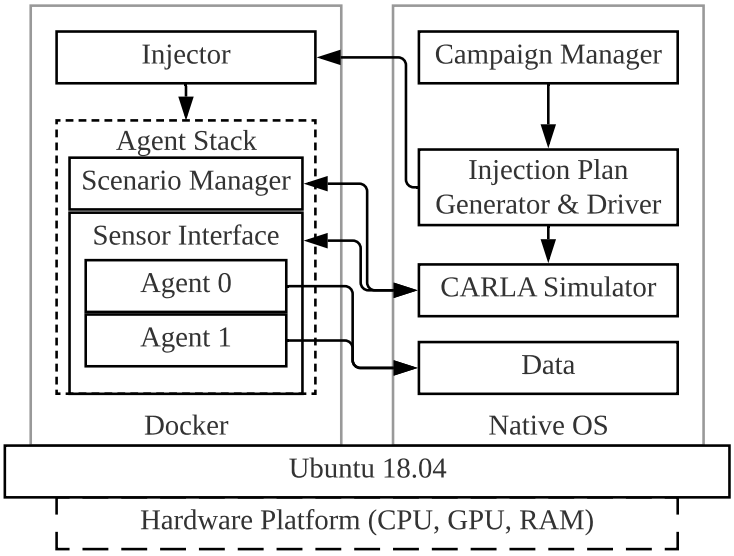


Figure 5.3: Simulation platform.

for each time-step. *Waypoints Tracker* along with the *Control Unit* uses the local waypoints and PID controller to produce actuation outputs at each time-step.

5.4.2 Simulation Platform

We cannot use real-world data (such as KITTI dataset [196]) to evaluate DiverseAV as the fault may impact the vehicle trajectory, and therefore, the subsequent data captured via the sensors. Thus, in this work, we use a world-simulator to simulate the driving scenarios. An overview of our world-simulation platform is shown in Fig. 5.3. We use CARLA 0.9.10 [140], an Unreal Engine-based simulator, to simulate complex and realistic 3D environments for autonomous driving. We ran the CARLA simulator in synchronous mode with all sensor data (from 3 front-facing cameras (facing left, center, and right) and GPS and IMU) posted at 40 Hz.

The DiverseAV-enabled ADS consists of two Sensorimotor agents, a sensor interface for communication with the simulator, and a scenario manager that manages the driving scenario. The two agents can be configured to run in round-robin mode (i.e., agents receive sensor data at alternating time-steps) , duplicate mode (i.e., both agents receive all sensor data), or single mode, in which only agent 0 is active.

Before the simulation is started, the Scenario Manager sends to the driving scenario to CARLA simulator. The Campaign Manager reads experiment configurations and launches the Injection Plan Generator that selects the injection site (CPU vs GPU), the fault model

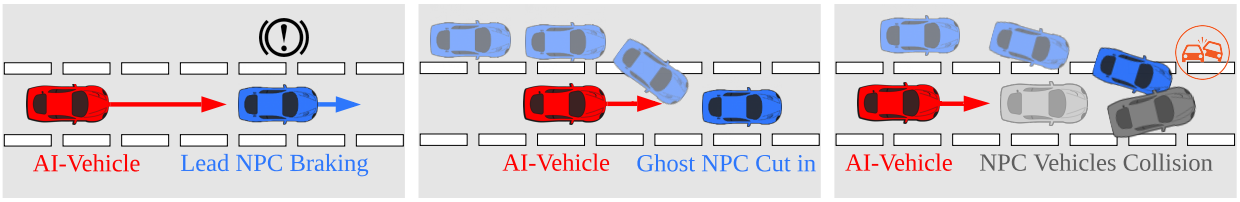


Figure 5.4: Driving scenarios. Left: lead slowdown. Middle: ghost cut in. Right: front accident. Red car: AI-vehicle, Blue Car: NPC-vehicle.

(transient vs permanent) and agent mode (single, duplicated or DiverseAV) for an experiment. The Driver invokes the simulator with the selected agent mode, and the selected fault injector.

5.4.3 Driving Scenarios

Safety-critical (Test) Scenarios

We created three safety-critical test scenarios, shown in Fig. 5.4. Scenarios of these kinds are considered high-risk by the National Highway Traffic Safety Administration (NHTSA), as stated in their pre-collision scenario topology report [197]. The safety-critical scenarios are about 30 seconds to 1 minute long and capture the most critical moments of autonomous driving. We used these scenarios in fault injection experiments to evaluate the effectiveness of error detection capabilities of DiverseAV.

Lead Slowdown: As shown in Fig. 5.4 (left), the vehicle (red), follows a leading NPC vehicle (blue), maintaining a distance of 25 meters. The NPC vehicle then performs emergency braking to slow down. The vehicle needs to recognize the situation and brake in time to avoid a collision. This is both a common and a high-risk scenario. Lead slowdown scenario is dangerous because it gives the follower vehicle little time to react, often resulting in a rear-end collision with the leading vehicle.

Ghost Cut in: As shown in Fig. 5.4 (middle), the vehicle (red) is driving on the road while maintaining speed, and an NPC vehicle (blue) approaches from the left adjacent lane. The NPC vehicle then cuts in front of the vehicle with a small longitudinal margin. The vehicle needs to reduce the throttle, slow down, and brake if necessary to avoid colliding with the side of the NPC vehicle. In this driving scenario, there is little to no warning prior to the cut-in maneuver of the NPC vehicle. This is especially dangerous for the vehicle as our agent does not use rear-end camera giving it less time to see the NPC vehicle and react to avoid collision.

Front Accident: As shown in Fig. 5.4 (right), the vehicle (red) is following a leading NPC vehicle (gray) in the same lane, and another NPC vehicle (blue) in the adjacent lane tries to merge but crashes into the leading NPC vehicle. Both vehicles' trajectories suddenly change because of the collision, and both vehicles stop. The vehicle needs to recognize this situation and stop in time to avoid an accident. Although this accident is rare, it is a high-risk situation because the vehicle might not recognize the abrupt change in positions and trajectories of the leading vehicle and outputs the wrong decision.

Long (Training) Scenarios

We constructed three long scenarios for training the error detector of the DiverseAV-enabled ADS. Our results, described in §5.5, show that the error detector parameters can be learned from these long driving scenarios with high precision and recall in detecting safety-critical faults. The long scenarios are based on selected routes from the 2020 CARLA Autonomous Driving Challenge, simulating normal, everyday driving tasks, such as vehicle following, lane keeping, turning, lane changing, and handling of intersections. We also enabled pseudo-random background traffic with a fixed random seed for each run. Each driving scenario simulation time is approximately 10–15 minutes long. The three long scenarios are based on Route02, Route15, and Route42, which are set in CARLA Town01, Town03, and Town06, respectively. These long scenarios require the AV to navigate in city and highway with dense traffic consisting of turns, intersections and traffic lights.

5.4.4 Fault Injection

We inject hardware faults by injecting architectural-level GPU or CPU errors that emulate consequences of underlying transient or permanent faults. In particular, we use `PinFI`[198, 199]) to inject faults into the CPUs, and `NVBitFI`[200]) to inject faults into the GPUs. Table 5.1 summarizes the results of the fault injection experiments on DiverseAV-enabled ADS.

GPU fault injections. We conduct following GPU FI experiments. (i) *Transient FI*: It is prohibitively expensive to inject all possible transient faults as the possible space of transient faults is extremely large (and equals to the number of dynamic instructions executed by the ADS). Therefore, we uniformly randomly selected 500 candidate dynamic instructions to transiently corrupt the destination register. (ii) *Permanent FI*: The ISA (Instruction Set Architecture) of the Titan Xp GPU includes 171 opcodes, and for each of the three

driving scenarios we perform fault injection for all 171 opcodes, with three repeated runs per opcode to capture any non-deterministic effects. Thus, resulting in 513 experimental runs per driving scenario.

CPU fault injections. We conduct following GPU FI experiments. (i) *Transient FI*: Similar to the GPU FI experiments, we uniformly randomly select 500 candidate dynamic instructions to transiently corrupt the destination register. (ii) *Permanent FI*: The Sensorimotor agent uses 131 Intel opcodes, and for each of the three driving scenarios we perform fault injection for all 131 opcodes, with three repeated runs per per opcode to capture any non-deterministic effects. We also perform injection of CPU faults using a modified version of `PINFI` to support a permanent fault model that is similar to the permanent fault model for NVBitFI, where all dynamic instances of a specified opcode are corrupted. Thus, resulting in 393 experimental runs per driving scenario.

In addition we run 50 experiments per scenario without fault as "golden" baseline runs. The golden runs serve as control experiments as the error-detector *must not* classify any of these runs as faulty. An error detector which falsely classifies a golden run as error-free will trigger frequent alarms and thereby, decrease the overall system availability.

5.4.5 Hardware Platform

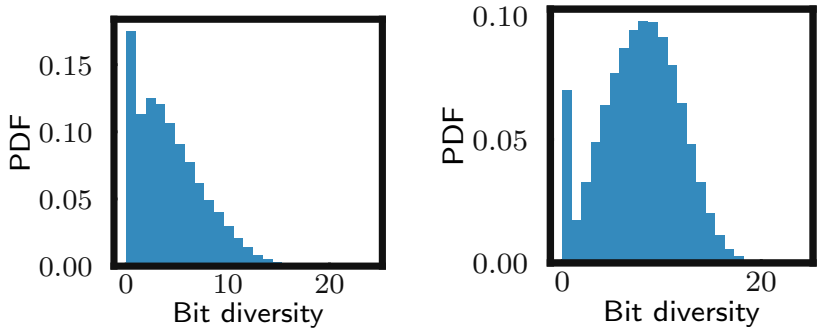
Our experimental setup uses XEON E5-2699v4 CPU with 64 GB of RAM and two Titan Xp GPU cards.

5.5 RESULTS

5.5.1 Characterizing Input Data Diversity

We characterize the diversity in sensor data between consecutive time-steps of autonomous driving on both simulated sensor data generated using the CARLA Simulator for our test driving scenarios [140] and KITTI dataset [196].

Fig. 5.5a shows the bit diversity–the number of bits difference in the 24-bit RGB color representation (8-bit per color), calculated per corresponding pixel location between two consecutive RGB camera frames. The distribution of bit diversity at the 50th percentile is 5 bits, and at the 90th percentile is 9 bits, out of the 24 bits of an RGB pixel. We also characterized the bit-diversity in camera images of a real-world dataset, KITTI, as shown in Fig. 5.5b. We found that the distribution of bit diversity for camera image data at the



(a) Simulator-generated data. (b) Real-world data.

Figure 5.5: Image pixel bit diversity.

50th percentile is 8 bits, and at the 90th percentile is 13 bits, out of the 24 bits of an RGB pixel. Our characterization shows that even though the semantics of the world does not change significantly from one time-step to another, the bit-representation of the world (captured via the cameras) changes significantly. The characterization holds for other sensors such as LIDAR, GPS and IMU, which we omit due to space constraints.

5.5.2 Characterizing the Impact of DiverseAV on Safety

Here we characterize the impact of the DiverseAV-enabled ADS on the vehicle's safety by evaluating the maximum divergence between the trace of the vehicle trajectory ($traj$) of an experimental run of a driving scenario generated using the DiverseAV-enabled ADS, and the baseline trajectory generated using the original ADS. The *vehicle trajectory* of an experimental run of a driving scenario is the trace of the path followed by the vehicle. Formally, it is a timestamped list containing the global position of the vehicle at any time t during the execution of that driving scenario, i.e., $traj = [pos_t | \forall t]$. We express the maximum divergence between a given trajectory ($traj^E$) and the baseline trajectory ($traj^B$) as $\delta_{pos}^{E,B}$, where $\delta_{pos}^{E,B} = max(traj^E - traj^B)$.

Fig. 5.6 shows the boxplot of $\delta_{pos}^{E,B}$ across three driving scenarios, calculated using 50 experimental runs (golden runs) of the scenarios. We characterize the divergence among the vehicle trajectories generated using the original ADS as well as the DiverseAV-enabled ADS. The baseline trajectory $traj^B$ used for calculating $\delta_{pos}^{E,B}$ for a given driving scenario was chosen as the mean of all the trajectories generated using the original ADS. Thus, the boxplots labeled "orig" show the distribution of the maximum divergence for the vehicle position across the experimental runs of a driving scenario when the vehicle was

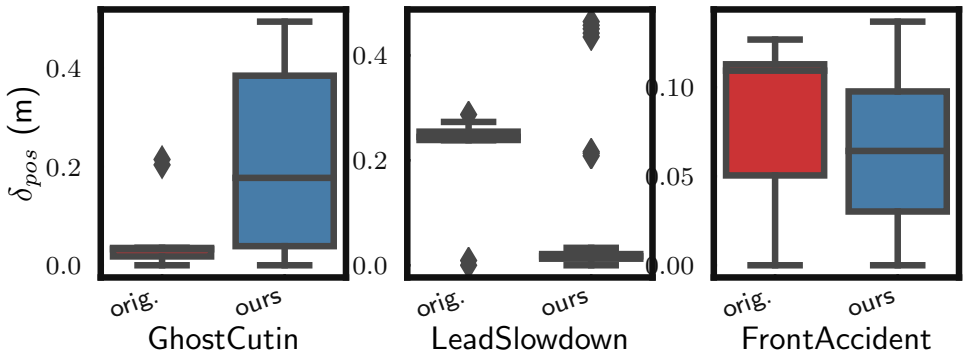


Figure 5.6: Impact on vehicle trajectory due to DiverseAV.

using the original ADS. Similarly, the boxplots labeled "ours" show the maximum divergence for the vehicle position among experimental runs when the vehicle was using the DiverseAV-enabled ADS with respect to the mean of the trajectories generated by the original ADS. Our characterization shows that the vehicle trajectories did not change significantly (<50 cm across all scenarios) when we used the DiverseAV-enabled system instead of the original ADS. Moreover, the DiverseAV-enabled vehicle neither experienced a collision nor broke any traffic laws in any of our experimental runs across the driving scenarios. Based on those observation, we conclude that our proposed design is safe and mimics the vehicle trajectory closely compared with the original ADS.

5.5.3 Characterizing Fault Propagation

Table 5.1 provides an overall summary of the experiments. Each row in the table shows the statistics for one fault injection (FI) campaign, which is characterized by a fault injection target and the driving scenario. In total, we executed twelve FI campaigns in which we injected faults into two targets (CPU and GPU) in three driving scenarios (LeadSlowDown, GhostCutin, and FrontAccident). We also used three additional training driving scenarios (Town01-Route02, Town03-Route15, and Town06-Route46) to train our error detector (not mentioned in the table). For each of the campaigns, we ran 50 golden runs (i.e., experimental runs without fault injections) to (i) characterize the simulation's nondeterminism, (ii) understand the impact of faults on the vehicle's safety, and (iii) test the error detector. Across all the FI campaigns, we quantify the safety of the vehicle in terms of accidents and trajectory violations. We marked an experimental run (E) as "trajectory violated" if $\delta_{pos}^{E,B} \geq 2.0$ (the maximum divergence between the trajectory of the experimental run (E) and the baseline run (B) is more than 2.0 meters). However, we varied the

Table 5.1: Summary of experimental runs in DUAL agent mode. DS: Driving scenarios (LSD - Lead Slowdown, GC - Ghost Cut in, FA - Front Accident scenarios); #active: # of FI experiments in which fault was successfully injected; #Traj Violations*: # of experiments with trajectory violation but without accident. #Acc.: # of experiments with accident.

FI Target	DS	#Active, Hang/Crash, Total FI	#Accidents	#Trajectory-Violations*
GPU-permanent	LSD	513, 83, 513	3	9
GPU-permanent	GC	513, 83, 513	14	2
GPU-permanent	FA	513, 81, 513	0	3
CPU-permanent	LSD	393, 287, 393	0	0
CPU-permanent	GC	393, 286, 393	0	0
CPU-permanent	FA	393, 287, 393	0	0
GPU-transient	LSD	500, 40, 500	0	2
GPU-transient	GC	500, 46, 500	0	2
GPU-transient	FA	500, 39, 500	2	0
CPU-transient	LSD	413, 171, 500	0	0
CPU-transient	GC	203, 70, 500	0	0
CPU-transient	FA	452, 199, 500	0	0

$\delta_{pos}^{E,B}$ parameter to reveal the detection capabilities of our proposed design. The trajectory of the baseline run is assumed to be the mean trajectory of all the golden runs.

Transient faults. Across all transient faults, CPU FI resulted in (i) highest percentage of hangs and crashes (41.2%; 440 out of 1068 runs[2]), and (ii) zero accidents and trajectory violations. A high percentage of hangs and crashes are expected for CPU FI campaigns because FI into CPU instructions is very likely to corrupt the program control flow or memory addresses, resulting in segmentation faults and broken pipes, among other problems. Hangs and crashes are automatically detected by the platform, thereby triggering the fail-back system which can bring the vehicle to a safe state. CPU FIs do not cause silent data corruption (SDC) because the Sensorimotor agent used in our work uses the GPU mostly for computations, whereas it uses the CPU for loading and setting the `Pytorch` program. Consequently, we observed a relatively low percentage of hangs and crashes for GPU transient faults (8.3%; 125 of 1500 runs). However, transient faults into GPU did lead to accidents and trajectory violations (0.4%; 6 out 1500 runs).

Permanent faults. We observe similar trends for permanent faults for CPUs and GPUs except for the fact that permanent fault resulted in significantly more hangs/crashes and

[2]The statistics is calculated by dividing total number of hangs and crashes in Table 5.1 in column 2 for CPU-transient faults and total number of fault activated experiments.

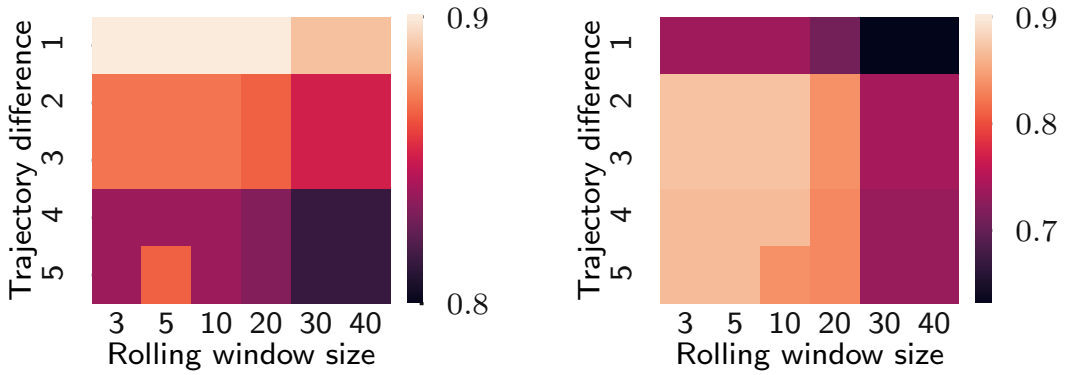
accidents/trajectory violations. CPU FI resulted in (i) the highest percentage of hangs and crashes (72.9%; 860 out of 1,179 runs), and (ii) zero trajectory violations and accidents. Similar to transient FI, we observed a relatively low percentage of hangs and crashes (16%; 247 out of 1,539 runs) in the case of GPU FI as compared with CPU FI, and a high percentage of accidents (1.1%; 17 out of 1,539 runs) and trajectory violations (with no accident) (0.9%; 14 out of 1,539 runs).

5.5.4 Characterizing Error Detection Capabilities

DiverseAV must be able to detect all safety-critical errors, i.e., faults that lead to a collision or significant trajectory divergence. In addition, it should not raise false alarms, especially for the golden runs (experimental runs of driving scenarios without fault injection). We evaluate error detection capabilities in terms of precision, recall, and lead detection time. Moreover, we parameterize the trajectory divergence using the parameter td. We mark an experiment as "trajectory violated" if $\delta_{pos}^{E,B} \geq td$, i.e., if the max difference between the experimental run of a driving scenario and the baseline trajectory exceeds td. This parameter impacts the number of cases that need to be detected by the DiverseAV. We evaluated DiverseAV's detection capabilities for $td = 1, 2, 3, 4, 5$ meters. The simulations of the driving scenarios have inbuilt non-determinism, and, as shown in Fig. 5.6, the natural variation in trajectory can be as high as 0.5 m; therefore, we chose $td > 0.5m$.

We evaluated the error detection capabilities of DiverseAV only for GPU faults, as all CPU faults either were detected by the platform (as hangs or crashes) or did not result in accidents or trajectory violations. We trained and tested DiverseAV on different scenarios to understand the generality of the proposed design. DiverseAV was trained using "long driving" scenarios and tested on safety-critical scenarios.

Fig. 5.7a and Fig. 5.7b show heat maps of the precision and recall values for our error detector across different parameters of td and rw (rolling window size). Overall, we found that the detector robustly detected the safety-critical faults and produced a low false positives rate across a range of parameters ($td \geq 2$ and $rw \leq 30$). The best performance (i.e., precision = 0.87 and recall = 0.87) was achieved with $td = 2$ and $rw = 3$. For these parameters, DiverseAV did not raise an alarm for any of the golden runs of the driving scenarios. Fig. 5.8 shows the lead detection time for the detector using the parameters $td = 2$ and $rw = 3$. The lead detection time is significantly higher than 1.0 second, allowing the fail-safe system ample time to take control and react to the driving situation at hand.



(a) Precision. (b) Recall.

Figure 5.7: Detecting safety-critical GPU faults.

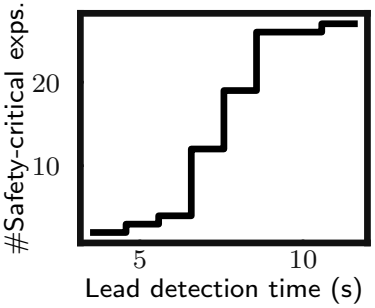


Figure 5.8: GPU FI lead detection time.

The above results indicate that our approach achieves high recall and precision in detecting runtime errors. In the context of an ADS, high recall and a corresponding low false-negative rate are important because they indicate that most faults are being detected or they do not affect the vehicle behavior. When an error is detected, the vehicle fails over to a backup system that brings the vehicle to the safe state.

5.5.5 Performance Overhead

The performance overhead characterization shows that DiverseAV increased compute utilization marginally and significantly increased (by $1.26\times$) memory utilization. That is expected because of the two agents employed in DiverseAV maintains their own internal (private) state. The GPU memory usage increased by 26% from 757 MB to 955 MB because of the additional agent instance, and the CPU memory usage increased by 19% from

Table 5.2: Average system resources used by single-agent, DiverseAV-enabled and fully duplicated (FD) ADS.

	CPU	GPU	RAM	VRAM
Single Agent	4%	14%	2,258 MB	757 MB
DiverseAV	5%	15%	2,689 MB	955 MB
Full duplication (FD)	8%	28%	4516 MB	1514 MB

2,258 MB to 2,689 MB because of the additional agent and sensor interface. However, DiverseAV significantly reduced the computational overhead compared to a completely duplicated system; 37.5% for CPU, 46.4% for GPU, and memory overhead by 39.5%; with no additional hardware requirements. A summary of performance overhead is shown in Table 5.2. The additional agent did not double GPU memory usage because part of the memory (about 559 MB) is reserved for Pytorch CUDA context. An additional agent increases memory usage by 200 MB because of additional weights and input/output tensors.

Although the compute resource utilization is low for the Sensorimotor agent used in this chapter, we know from our experience that compute utilization for a real-world AV is high, requiring multiple CPUs, GPUs, and FPGAs [194].

5.6 DISCUSSION

5.6.1 Errors missed by DiverseAV

DiverseAV can detect failures only when a fault impacts the redundant agents differently, thereby producing different actuation commands. However, there is a nonzero chance that the diverse agents will produce similar actuation outputs even in the presence of an error. However, in our experiments, we find the probability that a fault will result in similar actuation outputs in both agents and also result in a safety hazard to be small (0.001 for GPU faults; which is estimated using eq. (5.4)).

$$\text{missed safety hazard cases}/\text{total fault injection experiments} = 4/3189 \tag{5.4}$$

5.6.2 Comparison with Fully Duplicated ADS

We compared the accuracy of error detection capabilities of DiverseAV with the fully duplicated ADS (FD-ADS) (e.g., [182]). In this setup, the two redundant agents are ex-

ecuting on its own dedicated hardware. Similar to DiverseAV-enabled ADS, the two agents share the sensor; however, each agent receives the exact same data. Due to the limitation of the simulator we cannot run two agents concurrently in lock-step without significantly modifying the simulator (as well as the communication interface between the agent and the simulator). Therefore, we emulated the FD-ADS setup by executing single-agent setup twice for a given driving scenario; the first execution is injected with a transient or a permanent fault (thereby, emulating a fault-injected agent), and the second execution is used as a reference for comparison of actuation commands (thereby, emulating a non-faulty agent). The emulated setup is reasonable because the simulator is discrete, and therefore, each simulation of a driving scenario with the same driving scenario configuration parameter produces similar trace (i.e., trajectories and actuation outputs). Since, the trace of the experimental runs are not bit-by-bit match (even for the golden runs), we use a statistical model to detect errors. The error detector is trained using sliding-window-based approach discussed in §5.3.

Compared to DiverseAV, FD-ADS achieved precision of 0.18, recall of 0.84, and FPR (false positive rate) of 0.07 across 500 runs of each scenario (1500 total runs). FD-ADS correctly identified most cases of true positives (accidents and trajectory violations) but falsely identified significant number of fault-injected runs which did not lead to safety hazards as errors. Thus, resulting in lower precision (and lower availability) compared to DiverseAV-enabled ADS. The FD-ADS setup has low precision because (i) fully duplicated system is overly sensitive to mismatches between the control outputs, and (ii) we emulate the FD-ADS setup using single-agent system. Similar to the DiverseAV-enable ADS, none of the golden runs were marked in error in the FD-ADS setup.

5.6.3 Comparison with Single Agent ADS

We compared our model with the single agent system, in which the ADS is using *only* a single-agent to control the Ego vehicle. Both in the FD-ADS and DiverseAV-enabled ADS, the system is using two agents and hence, there is a reference available to us for comparing the outputs. However, in the single agent system, there is no reference available for comparison except for identifying temporal anomalies in the timeseries data.

It is difficult to identify errors using temporal outlier or range-based detectors [201] as occasional blips (that are within the acceptable output range) frequently occur in the actuation outputs (see Fig. 5.1(b)). Increasing the sliding-window size to smooth the outputs in order to remove blips reduces the overall recall of the error detector model, while decreasing the sliding-window size results in too many false positives. To illustrate the

difficulty in designing a temporal outlier-based error detector using a single agent, we developed a sliding window-based anomaly technique similar to the one used in this chapter. The best performance, in terms of F1-score, achieved by the single-agent system yields in precision and recall of 0.17 and 0.52 respectively; which is significantly smaller compared to both FD-ADS and DiverseAV-enabled ADS. We must note that it might be possible to detect safety-critical faults in a single-agent ADS; however, that approach will require large amount of data and complex machine-learning models such as LSTM/RNN [202] to train the detector. Our future work will explore such models. In contrast, DiverseAV is simple requiring black-box comparison of actuation commands with interpretable statistical model using few model parameters.

5.6.4 Impact on safety

Although our evaluation shows that DiverseAV does not impact the vehicle's safety, it is plausible that the proposed design may have safety implications in some critical driving scenarios. This is because the sensing frequency of each agent is reduced by half (and hence the available history also reduced by half) which may lead to higher uncertainty or delayed response. We do not quantify the increased uncertainty but empirically show that safety of the system is maintained across several driving scenarios. This is expected as commercial ADS have been shown to be safe even with an input data rate that is much lower than the nominal rate [203]. This robustness to the input data rate exists because commercial ADS include a significant engineering margin. For an ADS with lower engineering margins, the sensor data distribution can be adjusted so that some input data is sent to both agents, thus resulting in a input data rate reduction less than 50%, albeit at the expense of greater performance overhead. Our future work will include investigation of efficacy of DiverseAV on these critical driving scenarios with other data distribution strategies.

5.7 RELATED WORK

Safety-critical systems employ one or more of the following techniques to protect against faults.

Circuit hardening, which includes techniques for reducing the incidence rates by manufacturing process improvements or modifications of operational parameters, such as clock frequencies or voltages. However, because random hardware fault sources are often

external (e.g., high-energy neutrons, alpha particles, electromagnetic interference, voltage drops, or excessive heat), mitigation of the incidence rate is often a partial solution [204].

Hardware design modifications, which include error detection and correction at the circuit, micro-architecture, and architecture levels (e.g., instruction retry [205–207], ECC [208–210], checkers [211], and parity codes [175]). Significant effort has been devoted to hardware-level redundancy such as lockstep duplication [180–182], thread redundancy inside a single core [181, 183], or across cores [184–187], including partial redundancy techniques [212, 213]. However, those solutions require hardware support for thread synchronization, and incur significant area and power overheads. Furthermore, because of those overheads, applications of these techniques tend to be associated with larger arrays of circuit elements for which the overheads can be amortized. Thus, in typical chips, a significant portion of vulnerable elements are not protected (e.g., small SRAM arrays, flip-flops, compute units and pipelines) leading to silent-data corruptions [214, 36].

Software algorithms or enhancements, which include error detection and correction at the software level with negligible dedicated hardware support. Techniques include (i) algorithm-based error detection [215, 216], (ii) assertions [178, 179] (ii) monitoring [217–220]), and software-based redundancy. Software-based redundancy techniques includes instruction duplication and checking via compilation techniques [221–224], process-level duplication, and building on transactional memory [225], among other solutions [226]. Software enhancements usually incur lower overhead than hardware methods. However, the applicability of software techniques tends to be dependent on the specific target software, so significant portions of the software are often left unprotected. Moreover, identifying algorithms and methods that provide high-coverage is challenging and requires an in-depth understanding of the fault-propagation in the application [227].

Enforcing diversity helps to tackle common cause failures (CCF) such as design bugs and software implementation defects. Diversity can be enforced at the instruction and program-level [189], temporal-level (e.g., instruction-retry), design-level [188] and the data-level [190]. The assumption is that the diverse designs are susceptible to different faults and therefore, the outputs of the two diverse designs will significantly differ on encountering a systematic fault. However, design diversity is too costly (in terms of man-hours required to develop N-versions or data transformation techniques), and arguably challenging to enforce in practice.

Putting DiverseAV into perspective. DiverseAV is a lightweight, software-based redundancy technique that exploits the temporal data diversity present in the sensor data of dynamical autonomous systems to achieve high-coverage error detection for transient and permanent hardware faults without incurring significant computational overhead

(in terms of performance and hardware/software resources). Thus, enabling detection of safety-critical faults in the computational hardware elements of the entire ADS. In contrast to full hardware or software duplication, DiverseAV ensures data and (internal) state diversity between the two agents. Moreover, DiverseAV is a plug and play solution, and the engineering and development effort for enabling DiverseAV is small (unlike above-mentioned diversity techniques). To the best of our knowledge, there is no existing work on achieving ADS redundancy by leveraging temporal data diversity in sensors.

5.8 CONCLUSION

In this chapter, we proposed DiverseAV, a low-cost redundancy technique for autonomous driving agents that leverages temporal diversity for safety-critical error detection. Our results show that DiverseAV is highly accurate (in terms of precision and recall), and detects failures sufficiently far in advance.

In future, we plan to quantify the uncertainty introduced by our DiverseAV design as well as extend the model to other autonomous systems such as unmanned aerial vehicles. We also plan to extend the framework to help localize the faults to enable more fine-grained mitigation instead of handling of all faults using a fail-back system.

CHAPTER 6: AV: WATCH OUT FOR THE RISKY ACTORS: IDENTIFYING IMPORTANT ACTORS IN DYNAMIC ENVIRONMENTS FOR SAFE DRIVING

Driving in a dynamic environment with other actors is inherently a risky task, as each actor influences the driving decision and may significantly limit the number of choices in terms of navigation and safety plan. The risk encountered by the Ego actor depends on the driving scenario and the uncertainty associated with predicting the future trajectories of the other actors (NPCs) in the driving scenario. However, not all NPCs pose a similar risk. Depending on the NPC's type, trajectory, position, and the uncertainty associated with these quantities, some NPCs pose a much higher risk than others. The higher the risk associated with an NPC, the more attention must be directed towards that NPC in terms of resources and safety planning. In this chapter, we propose a safety importance metric (SIM) that captures the importance of each NPC in the world with respect to their ability to create a safety hazard. In particular, the SIM characterizes the decrease in the Ego actor's driving flexibility with respect to a given NPC or a driving scenario. The more constrained the Ego actor the higher the chance of a safety hazard, and therefore, the higher the risk. By characterizing a real-world dataset using our metric, we find that $<$0.1% of NPCs in the environment constrain the Ego actor. We propose a novel neural-network-based model to estimate the *importance* metric at runtime with significantly less overhead in terms of computation and memory, while meeting the deadline requirements for runtime monitoring. Moreover, we show that integrating SIM with the offline assessment techniques eliminates the need to test the adverse effect of faults or attacks for all NPCs, as only few NPCs are important at any given time; thus, we reduce the test set and provide up to 24× acceleration over the current state-of-the-art assessment techniques.

6.1 INTRODUCTION

Driving in a real-world environment with ever-changing dynamics and among other actors is inherently a risky task. Each actor in the environment can significantly influence the driving decisions and limit the number of choices available in terms of navigation and safety plans. In extreme cases, any of these actors can, willingly or unwillingly, thwart the driver in safely completing the driving task. Thus, it is critical to quantify the risk posed by each actor in terms of their ability to create a safety hazard.

Human drivers using their perception and prior knowledge implicitly and continuously assess the risk associated with both the driving scenario and the other actors. Through this assessment, human drivers identify the most important actors in the envi-

ronment and focus their attention on those actors to prevent safety hazards. However, a significant number of accidents occurs due to human negligence or inability of the human drivers to accurately assess the importance of each actor. This can happen when (i) driving under the influence or otherwise distracted, (ii) the driver is less than fully skilled, and (iii) expectations mismatch because an actor violated the traffic rule or made a sudden action (e.g., braking) [228]. Artificial intelligence (AI)-based self-driving vehicles promise to eliminate these causes of accidents by (i) removing humans from the driving loop and (ii) making driving decisions by being cognizant of all the actors in the environment. However, AI-driven autonomous systems, despite their successful demonstrations, have failed to deliver on this promise [3, 2], as is evident from the significant deployment delays. The strategy of paying equal attention and taking preventive actions with respect to all actors works well for most driving scenarios. However, in rare situations, when there is a need to rapidly respond to an adverse event (such as an actor behaving erratically and closely cutting into the Ego lane[1]), the Ego actor cannot afford to spend its resources in identifying, tracking, and mitigating the risk associated with every actor on the road. Instead, it must identify and track the most important actor(s) and take a mitigating action that minimizes the risk and, therefore, the probability of hazardous outcomes. Hence, it is critical for designers to learn from humans and develop techniques for assessing the importance of each actor in terms of their ability to create a safety hazard and then mitigating those hazards at runtime.

In this chapter, we design and develop a novel *safety importance metric* (SIM) to characterize the threat posed by an actor by estimating its influence on the Ego actor's decision process in a given environment, which in turn is estimated in terms of reduction in driving flexibility. SIM depends on: (i) the environment (i.e., driving scenario) and (ii) the uncertainty associated with determining the driver's future trajectories.

Driving scenario. Driving scenario is the specification of the initial location of the Ego actor, the map, and the trajectories of NPCs. With each additional NPC on the road, diverse safe trajectories that can be followed by the Ego actor may be significantly reduced. This decrease in safe trajectories constrains the Ego actor in its decision making process, and it significantly reduces the failback options available to the Ego agent in case of unforeseen situations. Hence, decrease in diverse safe trajectories is analogous to the *importance* of an actor because it requires more attentive driving. This is illustrated in Fig. 6.1.

Uncertainty. Importance of an actor also increases if the future trajectory of the NPCs

[1]Ego actor is the actor that is being tested and is under the control of the tester. Ego lane is the lane on which the Ego actor is driving.

cannot be predicted accurately by the Ego actor. The accuracy of prediction decreases due to unclear intention or sudden change in intention of other participants in the driving environment (e.g., pedestrian decides to suddenly stop or run, emergency braking or breakdown of other vehicles), measurement noise (e.g., due to a bad weather), or imperfections in ML/AI models (e.g., pedestrian is not detected). This is illustrated in Fig. 6.2.

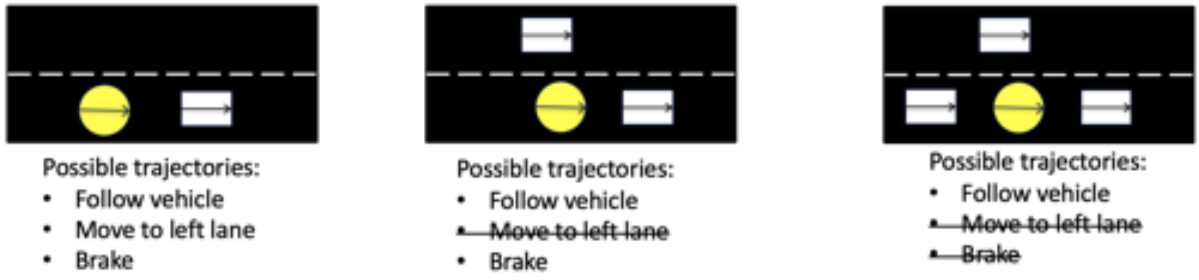


Figure 6.1: Driving scenario complexity increases from left to right as number of possible future paths/actions decreases. Each scenario is safe if all actors follow "duty of care." However, there is inherent risk that one or more can behave erratically; hence the decrease in number of actions/choices is analogous to increase in *importance* of an actor. Yellow circle denotes Ego actor and white rectangular boxes represent other actors.

Figure 6.2: Ego actor is more constrained from left to right as there is higher uncertainty in trajectory estimates of one or more actors' states. Increase in uncertainty decreases the possible future choices/action that can be taken, thereby increasing the risk. Thus, actors that constrain the Ego actor the most are the most important actors in the environment. However, increase in uncertainty of an actor that is far away may not have any consequence with respect to the driving decision; hence, increase in uncertainty only matters if it influences the Ego actor's driving decision.

The contributions of this chapter are as follows:

(a) **Safety importance metric formalization**: We define a new metric, safety importance metric (or SIM), to quantify the importance of each actor in the environment as well as the environment itself. The SIM score for each actor and driving scene

is normalized between zero and one. Thus, the higher the SIM, the higher the risk. Our importance metric is motivated from Barlow's importance metric [229]. Barlow's importance metric measures the conditional probability that the system failure is caused by (i.e., coincides with) the failure of a given component; thus, enabling system designers to calculate the relative importance of each component for a static system. Barlow's importance metric requires fault-tree specification of the system and serves as a useful guide during the system development phase as to which components should receive more urgent attention in achieving system reliability growth.

In contrast, SIM is designed for online importance assessment for a dynamically evolving system consisting of an Ego actor and other actors in the environment. Fault trees for such systems do not exist. Hence, SIM uses path-planning algorithms [230] to capture the relationship between the Ego actor's driving decision and other actors' states. Path-planning algorithms find one or more collision-free driving trajectories from the current location to the goal location. Conceptually, we use these path-planning algorithms to calculate SIM as the ratio between (i) decrease in the number of drivable trajectories (i.e., decrease in driving flexibility) in the presence of an actor and (ii) total number of drivable trajectories. SIM allows us to identify the relative importance of each actor. The calculation of SIM depends on both the state of the Ego actor and the state of the other actors in the environment. Therefore, we model the uncertainty in SIM by modeling the uncertainty associated with the state of the actors in the environment, including the Ego actor. We report both mean and variance associated with this metric.

(b) **NN-based SIM evaluation for rapid monitoring.** Evaluating importance at runtime, which meets latency deadlines, is difficult because the evaluation depends on the monitoring system's ability to find all future possible driving trajectories, which is computationally intractable. To address this challenge, instead of identifying *all future drivable trajectories* from current location to the goal location, we identify all reachable locations in the vicinity of the Ego actor's current location.[2] SIM can use the above-mentioned path-planning algorithms to find a path from current location to a vicinity location. We assume that the Ego actor uses a hierarchical planner consisting of a high-level planner (which makes the routing decisions from source to destination) and a low-level planner (which makes the local decisions such as steer-

[2]We can divide the map into a grid of fixed-size cells. In this abstraction, the driving flexibility can be quantified as the total number of reachable cells from the current location in a given fixed time interval.

ing, braking, and throttling). Calculating reachability using above-mentioned path-planning algorithms is still prohibitively expensive for online assessment. Hence, we use a neural network (NN) that takes bird's eye view as the input and outputs a binary image in which all reachable vicinity locations along the route that are reachable in a given fixed time interval are marked as one, and non-reachable locations are marked as zero. NN-based implementation can be executed at runtime with low overhead while meeting deadline constraints. *NN-based implementation is 14.5× faster when compared with RRT*-based [230] implementation. Overall execution time is significantly less compared to the 180 ms threshold set by ISO 26262 [97] for monitoring of adverse events (such as fault detection and adversarial actors, among others).*

(c) **TestScenarios benchmark consisting of important scenarios and actors**: We evaluated the proposed metric on nuScenes driving dataset [231]. The nuScenes dataset consists of 1000 annotated driving scenes, each 20 seconds long, that are taken from busy local roads in Boston and Singapore. It consists of 1.4M camera images. We use this dataset because (i) it is openly available, (ii) it provides APIs to extract map and bird's eye views for the camera images, and (iii) most importantly, it has significantly more labels than any other dataset. For example, it contains 7×more object labels than KITTI dataset [196]. *Our evaluation shows that <1% of actors are important (i.e., importance score of >0.9). Similarly, our evaluation shows than only 1.4% of the driving scenes are important.* Using nuScenes dataset, we curated a new benchmark, TestScenarios, which consists of inherently hard-to-navigate driving scenarios and important actors. We assert that developers can use the TestScenario benchmark to significantly boost development, testing, and deployment of autonomous vehicles on the road.

(d) **Proactively mitigating safety hazards.** The proposed importance metric allows us to mitigate safety hazards proactively. We develop a safety engine that (i) monitors the importance of each actor at runtime and (ii) disengages the Ego agent sufficiently in advance to mitigate the potential safety hazard safely. The safety engine uses a threshold-based intervention strategy in which the Ego-agent is disengaged when the sum of predicted importance of all actors in a driving scenario increases significantly. The safety engine prevented an accident by disengaging the Ego agent in four out of five driving scenarios. In these scenarios, without the safety engine, the Ego agent's driving decisions lead to accidents.

(e) **Accelerating offline testing and assessment.** The proposed importance metric al-

lows us to further accelerate the assessment techniques such as *Bayesian Fault Injection* (BFI) that was proposed in Chapter 3. BFI gives equal importance all actors in the scene when estimating the probability of collision under the influence of a fault. However, as discussed above, only a few actors ($\sim 0.7\%$) are important while driving. Hence, we integrated the importance analysis with BFI. Importance-driven BFI accelerates the fault assessment by 24× when compared to vanilla BFI. Note that this is an additional 24× acceleration over 3690× acceleration that BFI provides compared to existing fault injection techniques.

Although, in this chapter, we focus only on self-driving cars as a use case, our framework is general and can be applied to other navigation-based autonomous systems such as unmanned aerial vehicles and drones.

Related work. There are mainly two lines of research associated with defensive driving: (i) identifying safe distance from other actors assuming everyone follows the "duty-of-care" policies, (ii) identifying out-of-training-distribution (OOD) scenarios so as to plan for the worst case for avoiding collisions. *Duty-of-care approaches* such as Safety Force Field (SFF) [232] and Responsibility-Sensitive Safety (RSS) [233] are geared towards estimating the safe distance from other actors assuming that everyone follows rules of the road. Additionally, the goal is to identify the culprit in case of a safety hazard. *OOD* models such as [234] are geared towards identifying driving scenarios in which the Ego actor's future trajectory (i.e., plan) has significant variance. Such variance can be characterized by using an ensemble of models or diverse data. Under high variance, the Ego actor chooses to use the most pessimistic plan in order to avoid collisions. For example, in [234], the Ego actor's plan is generated using Bayesian imitative model, and the variance of the imitation prior with respect to the model posterior is used as a proxy for identifying distribution shifts. On detecting a distribution shift, the Ego actor can either plan for the worst-case model or the average model. Finally, in another line of research [235], authors quantify the impact of uncertainty in driving performance, but do not quantify the quality of the generated plan in terms of its safety. In contrast, our goals are to (i) identify the most important actors by characterizing their negative influence on the degree of freedom of the Ego actor's navigational choices, (ii) quantify the SIM of the current plan and proactively mitigate safety hazards.

6.2 FORMALIZING AND QUANTIFYING SAFETY IMPORTANCE METRIC

The current approaches in identifying safety-critical actors on the road rely *only* on forward-simulating techniques to identify collision sets/trajectories. However, these tech-

niques do not account for the attention required for driving in strenuous driving scenarios where the Ego actor has limited number of choices/plans.

The goal of our work is to quantify the importance of each actor for a given driving scenario. To that end, we propose a novel metric that resembles a human driver's intuition and reasoning in safe driving. In this section, we formalize the importance metric under the following assumptions:

Assumption 6.1. Autonomous driving system design — In keeping with the industry standards, we assume that the overall system must consists of sensors, object detection, trajectory prediction, planner, and controller. We also assume that the system consists of enough computational resources to perform all computations within the specified time.

Assumption 6.2. Perception system — We assume that the perception system is able to detect all actors that are within some distance threshold d. However, the measured state of the object such as bounding boxes or positions are noisy. We do not consider issues of: (i) *fragmentation* caused by model's inability to match the detected actor to its trajectory, and (ii) *false detection* that lead to appearance of ghost objects in the world as perceived by the Ego actor.

Assumption 6.3. Global routing system — We assume access to a global navigation system that we can be used to specify high- level goal locations and the availability of routes to reach to that goal location.

Assumption 6.4. Inverse dynamics — We assume access to an inverse dynamics model (PID controller, $\mathbb{I}$) that performs the low-level control – inverse planning – a_t (i.e., steering, braking and throttling) provided the current and next states (i.e., positions) s_t, and s_{t+1}, respectively.

6.2.1 Determining the importance of an actor

In our abstract model, importance is characterized in terms of decrease in driving flexibility. Driving flexibility is characterized by the number of unique driving trajectories (or actions) that are available to the driver at runtime. The higher the driving flexibility, the lower the importance (and risk) because the Ego actor's decision depends less on another actor's decisions. We estimate the importance for all the neighboring actors individually.

In this section, we first formalize the importance metric for the oracle setting in which the current and future states of the actors are known with no uncertainty. We can extract these states from benchmark datasets where ground truth labels are available. Finally, we

extend the definition of importance to incorporate noise to enable real-world monitoring of actors. In the real world, the current and future states of the actors are estimated at runtime and therefore may have significant uncertainty.

6.2.2 Determining importance with ground truth data

Let us assume that there are N actors in the world including the Ego actor. Let us denote the state of an actor i at time t by $x_t^{(i)} \in \mathbb{R}^3$, and the trajectory (i.e., trace of the actor's state) from time t to $t+k$ given by $X_{t,k}^{(i)}$. Let us denote the trajectory of all actors except the Ego actor from time t to $t+k$ by $\mathbb{X}_{t,k} = X_{t,k}^{(1)}, ..., X_{t,k}^{(N-1)}$. We can now describe the driving scenario ($\mathbb{S}$) from time $t=0$ to $t=T$ as a tuple consisting of a map ($\mathbb{M}$), trajectories of all the actors except the Ego actor, and the initial position of the Ego actor ($x_{t=0}^{ego}$):

$$\mathbb{S} =< \mathbb{M}, \mathbb{X}_{0,T}, x_{t=0}^{ego} > \tag{6.1}$$

For now, let us assume that we have access to an oracle local planner (f_p) that uses the trajectories of all the other actors from time t to $t+k$ ($\mathbb{X}_{t,k}$) and the position of the Ego actor at time t (x_t^{ego}) to generate a set of future trajectories ($\mathbb{Z}_{t,k}^{ego}$) that the Ego actor can follow safely from time t to $t+k$ while following all the rules of the road. The design and implementation of the local planner f_p is discussed later in §6.3.

$$Z_{t,k} = f_p(\mathbb{M}, \mathbb{X}_{t,k}, x_t^{ego}) \tag{6.2}$$

The set consisting of all the navigable future trajectories in the absence of all actors is given by eq. (6.3).

$$Z_{t,k}^{\emptyset} = f_p(\mathbb{M}, \emptyset, x_t^{ego}) \tag{6.3}$$

Similarly, the set consisting of all navigable future trajectories in the absence of i^{th} actor is given by eq. (6.4). Here, $\mathbb{X}^{/i}\}_{t,k}$ contains trajectory of all actors except the i^{th} actor.

$$Z_{t,k}^{/i} = f_p(\mathbb{M}, \mathbb{X}_{t,k}^{/i}, x_t^{ego}) \tag{6.4}$$

We can now define the importance of a specific actor i from time t to $t+k$ as:

$$\rho_{t,k}^{(i)} = \frac{Z_{t,k}^{/i} - Z_{t,k}}{Z_{t,k}^{\emptyset}} \tag{6.5}$$

The normalization constant $Z_{t,k}^{\emptyset}$ enables us to compare the importance across different

driving scenarios.

We can similarly define the total importance ($\rho_{t,k}$) of a driving scenario as the normalized reduction in future trajectories due to presence of all actors on the road.

$$\rho_{t,k} = \frac{Z^{\emptyset}_{t,k} - Z_{t,k}}{Z^{\emptyset}_{t,k}} \tag{6.6}$$

6.2.3 Determining importance with noisy measurements

Now we can consider the case of noisy measurements. In this setting, the Ego actor measures the current state of an actor using its sensors denoted by $o^{(i)}_t$ and uses past and current measurements to estimate the current state $\bar{x}^{(i)}_t$. $\bar{x}^{(i)}_t$ is different from $x^{(i)}_t$ (defined in previous subsection), as $\bar{x}^{(i)}_t$ captures the uncertainty/noise associated with the state. The horizontal bar on the top of the symbols used here is to distinguish between the noisy data and ground data. Typically, $o^{(i)}_t$ is the bounding box that is detected using object-detection algorithms, such as YOLO [154]. The future trajectory of an actor i from time t to $t+k$ can then be modeled by the joint distribution as given by eq. (6.7).

$$\bar{X}^{(i)}_{t,k} = p(\bar{x}^{(i)}_t, ..., \bar{x}^{(i)}_{t+k} | o^{(1)}_1, ..., o^{(N)}_t) \tag{6.7}$$

Let $\mathbb{Q}_{t,k}$ denote the set consisting of sample of future trajectories for all actors, i.e., $\mathbb{Q}_{t,k} = \{q^{(1)}_{t,k} \sim \bar{X}^{(1)}_{t,k}, ..., q^{(N)}_{t,k} \sim \bar{X}^{(N)}_{t,k}\}$. As earlier, let us assume that we have access to an oracle planner $\bar{f}_p$ that generates all possible future trajectories that the Ego actor can follow safely given $\mathbb{Q}_{t,k}$, $\mathbb{M}$, and x^{ego}_t as given in eq. (6.8).

$$\bar{Z}_{t,k} = \bar{f}_p(\mathbb{M}, \mathbb{Q}_{t,k}, x^{ego}_t) \tag{6.8}$$

However, for different samples of $\bar{X}_{t,k}$ $\bar{Z}_{t,k}$ will be different. Using those samples, we can estimate the uncertainty in importance of an actor ($\rho^{(i)}_{t,k}$) and driving scenario ($\rho_{t,k}$) as discussed in the previous section.

6.3 DESIGN AND IMPLEMENTATION

Estimating importance as outlined above is difficult at runtime, as path-planning is PSPACE-hard, which is an indication of the computational intractability in the degrees of freedom of the agent [236, 237]. To address this challenge, we develop a novel domain- and data-driven approximation technique that meets the stringent latency and compute

requirements for runtime monitoring. The approximation is based on following design principles:

(i) *Exploring local routing decisions given the global route.* The proposed technique limits the exploration of alternate actions/driving trajectories within a fixed threshold distance d from the current Ego location. In other words, we assume that there is a global routing system (see **Assumption 4**) and the Ego actor can choose to deviate from the global route locally. Humans tend to drive in a similar fashion where they approximately follow the navigation system (such as Google maps) to reach the destination; however, several choices such as using a particular lane or vehicular speed is left to the driver's own discretion.

(ii) *Calculating reachability instead of driving trajectories.* The goal of our work is to calculate the importance of an actor or a driving scenario. In our abstraction, we estimate this by estimating the extent of decrease in choice of flexibility of driving, which in turn, is calculated by estimating total number of possible driving trajectories. However, the set of future trajectories is countably infinite and calculating future trajectories is computationally intractable. Therefore, instead of calculating the aforementioned set of driving trajectories, we calculate reachability of the Ego actor from its current location to several goal locations within the threshold distance d. These goal locations are the individual cells within the threshold distance d in the grid view of the map[3] as shown in Fig. 6.3. The reachability of the Ego actor from current location in the map to several goal locations is computed using a local planner. Thus, in our implementation, f_p is the reachability model and its outputs are used to compute the importance. We assert that total reachable destination locations (i.e., cells in the map) from the current Ego location serve as an approximate proxy for evaluating flexibility of driving; thereby, allowing us to approximate importance of each actor and the driving scenario.

6.3.1 Designing planner f_p

We use above insights to design f_p. The proposed f_p first discretizes the map into a grid of predefined, fixed size cells as shown in Fig. 6.3. Then it calculates the reachability of the Ego actor from the current location (*src*) in the grid to all destination cells (*dst*). The *dst* cells are all the cells within some threshold distance d. Each reachable cell in the grid

[3]We discretize the map into a grid of cells

contributes to the total degree of freedom; the higher the degree of freedom, the lower the importance. Fig. 6.4 conceptualizes this idea.

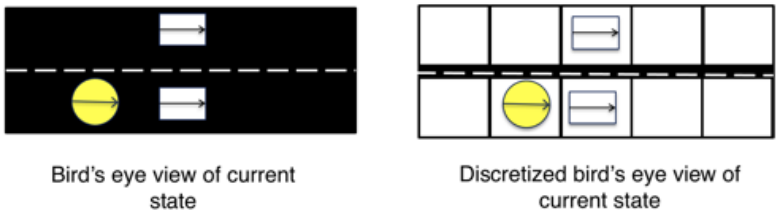


Figure 6.3: Discretization of the map into a grid of predefined fixed-sized cells.

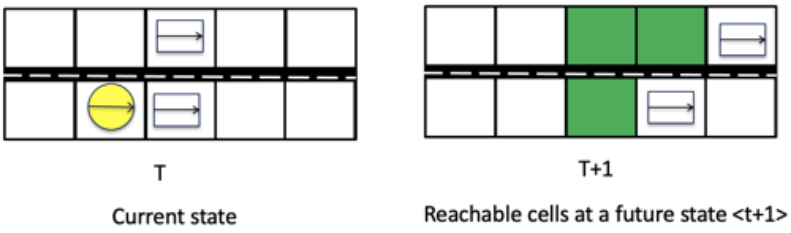


Figure 6.4: Using f_p, Ego actor can calculate all the safely reachable cells in the grid at time <t+1>. These reachable cells are shaded in green.

Though it is possible to compute reachable destination cells from the current location using graph search algorithms such as depth-first or breadth-first search algorithms. However, such an implementation will ignore temporal dynamics such as kinematics. Therefore, our implementation f_p uses RRT* [230] to find all reachable cells.

6.3.2 Approximating f_p using a neural network

Reachability calculation using RRT* is still not favorable due to its computational complexity and tail runtime behavior. Therefore, we further approximate the reachability computation by approximating f_p using a neural network. The inputs to the neural network consist of the bird's eye views (BEVs) of the camera data for θ epochs, i.e., BEV camera data from time $t-\theta$ to t. The output of the model is an indicator vector indicating all the reachable cells from time t to $t+k$. The indicator vector is of fixed size. Each cell in the grid corresponds to a particular index in the indicator vector. We formulate the reachability problem as a classification problem, where the goal of the neural network is to classify each cell in the grid (or the indicator vector) as reachable vs not reachable.

Remark 6.1. Note that the size of the grid up to distance d from the Ego actor depends on the map. For example, a single-lane road versus a four-lane road will have a different

number of cells within distance threshold d. However, such variability is not favorable because the output of the neural network must of be fixed size. Thus, in our implementation we always assume that the Ego actor is traveling on a three-lane road allowing it move left, right, or straight. This assumption is reasonable as the chance that the Ego actor crosses more than one lane in k time epochs is negligible if k is reasonably small.

Remark 6.2. Note that reachability datasets do not exist. Thus, we must create such a dataset using RRT* as described above. The goal of the neural network is to mimic the output of RRT*-based reachability model by classifying each cell in the grid as reachable versus not reachable. The neural network uses cross-entropy loss for this classification problem. Cross-entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. Moreover, we use a Bayesian Neural Network (BNN) [238] to model the uncertainty in trajectories of actors.

Finally, we estimate the importance using eq. (6.5) and the trained NN-driven f_p. Fig. 6.5 showcases the calculation of importance without considering the uncertainty. Yellow circle is the Ego actor and white rectangles are the other actors on the road. Arrows indicate the moving direction of the actors. Left figures show the state of the actors at time step t, and right figures show the reachable destination cells. Fig. 6.5a shows the bird's eye view of the driving scenario. Fig. 6.5b shows the reachable destination cells without deleting any actors in green and with deleting all actors in blue. Fig. 6.5c and Fig. 6.5d show reachable destination cells when deleting the bottom and the top actor, respectively. Clearly, the bottom actor reduces the flexibility of driving more than the top actor. Hence, the bottom actor is more important than the top actor. Here the reachable cells are calculated using the NN-based reachability model.

6.3.3 Alternate design choices

In our proposed formalism, we use reachable cells as a proxy for flexibility in driving in a driving scenario and use that to calculate importance. However, it is possible to use other proxies for capturing flexibility of driving:

(a) *Extent of change in the behavioral options:* Instead of defining choices in terms of reachable cells, one can also formulate choices in terms of behavioral choices, such as 'move to left lane', 'move to right lane', 'cruise', 'accelerate' and 'brake'. We can calculate the total number of choices with and without an actor to estimate relative

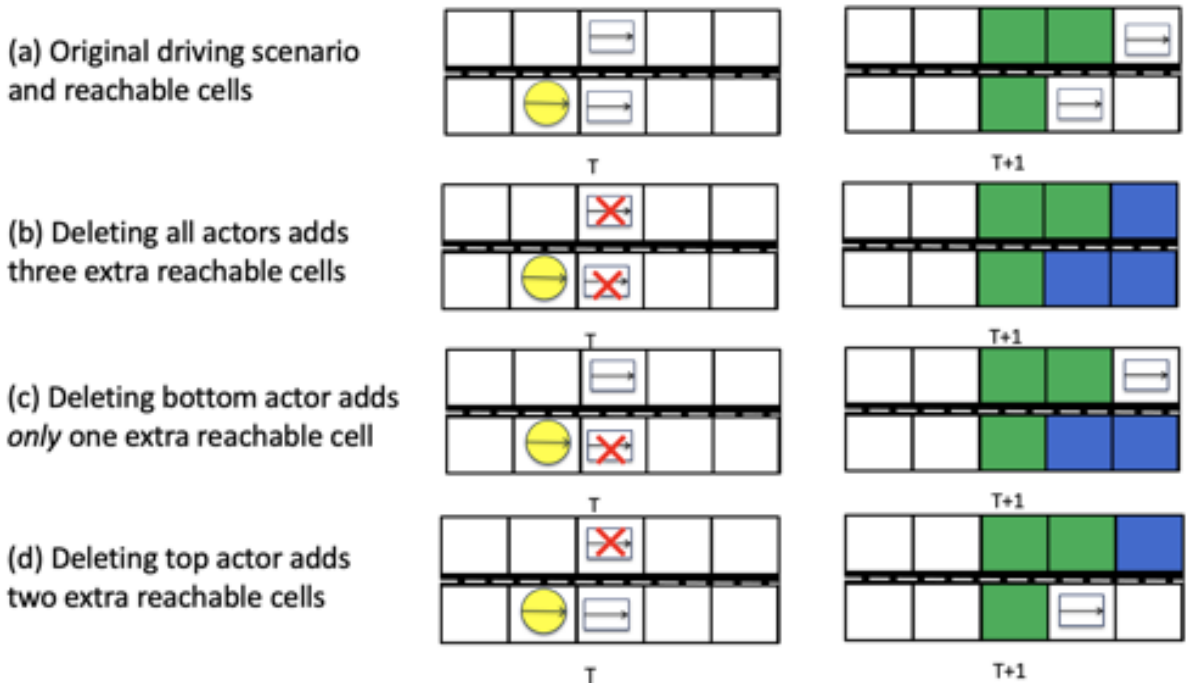


Figure 6.5: Depicting reachability and importance calculation. Green cells are the reachable cells before deleting any actor, and blue cells are additional reachable cells after deleting one or more actors. Deleting all actors allows us to calculate the total importance for a driving scenario, as shown is shown in (b). The total importance of the showcased driving scenario is 0.5 (estimated using eq. (6.6)). Using eq. (6.5), we can estimate the importance of each actor. Here the bottom actor is more important than the top actor because the number of reachable cells decreases more with respect to the bottom actor than with respect to the top actor.

importance. However, these behavioral choices can only be computed after estimating all the reachable cells. Hence, it is more computationally expensive than our proposed technique and therefore not suitable for online monitoring.

(b) *Extent of change in the planned trajectory:* Planning algorithms such as RRT*, as implemented in Pylot [239], find only one usable future trajectory. In this setting, importance can be approximated by calculating the difference between the planned trajectory with and without an actor. However, such approximation only calculates the influence of the actor on the driving decision without characterizing overall importance. For example, the difference between the planned trajectory with and without actor can be significant if there is only one non-player character (NPC) actor and that NPC is in front of the Ego actor. However, on a three-lane highway, despite the presence of the NPC, the Ego actor has several other choices, hence the

driving scenario/actor is less important (see Fig. 6.1 for illustration).

6.4 RESULTS

Datasets. We validate our proposed evaluation metric on the nuScenes dataset [231]. nuScenes consists of 1000 annotated driving scenes each of length 20 seconds, that are taken from busy local roads in Boston and Singapore. Ground truth 3D object labels are provided at 2 hz for objects that fall into 10 object classes including cars, trucks, pedestrians, and road barriers. The dataset contains 1.4M camera images, 390k LIDAR sweeps, 1.4M RADAR sweeps, and 7×more object labels than KITTI [196].

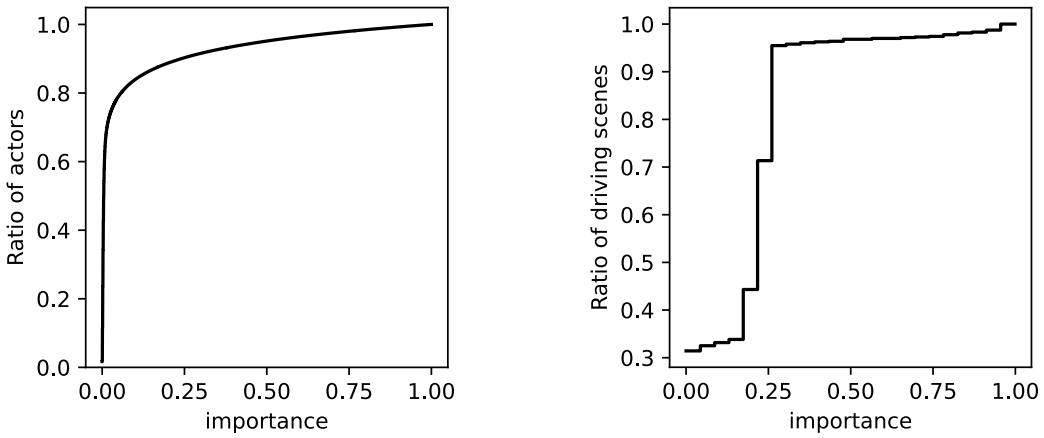
Performance Overhead. We characterize the overhead of our proposed technique both in terms of latency and memory requirements. Time to calculate importance for all actors within some threshold distance d depends on the number of actors that are present on the road. Thus, calculation of importance metric can exhibit significant tail behavior. The calculation of importance metric also depends on the underlying implementation of the reachability model. Table 6.1 shows the latency and memory overhead for calculating importance using RRT*-based and NN-based reachability model. NN-based reachability model 14.5×faster than RRT*-based reachability model in terms of latency. The time taken by NN-based implementation is significantly less compared to 180 ms threshold set by ISO 26262 [97] for detecting adverse events.

Table 6.1: Resource overhead.

Reachability model	**Latency** (mean $\pm$ std. dev)	**Memory overhead**
RRT*-based	1000 ± 320 ms	180 MB
NN-based	69 ± 46 ms	300 MB

Accuracy. We evaluate the accuracy of the NN-based reachability model using the nuScenes dataset. Precision and recall of the NN-model in imitating RRT*-based reachability model is 0.91 and 0.8 respectively. We also estimate the downstream effect of using NN-based reachability model on estimating the relative importance of actors by calculating the percent of entries that are differently ordered from RRT*-based importance estimation model. Our results show that only in 8% of the driving scenes[4] the relative importance ordering of actors is different from RRT*-based importance estimation model.

[4]A driving scene at time t is state of the world at any time step t. In our analysis, a driving scene is simply the camera image at time t.



(a) Characterizing actors

(b) Characterizing driving scenes

Figure 6.6: Characterizing safety importance metric on nuScenes dataset.

Evaluating importance metric on nuScenes dataset. We estimate the importance of (i) each actor in a driving scenarios and (ii) relative importance of driving scenarios across the nuScenes dataset. In this evaluation, we only characterize the mean importance. Though there is uncertainty in importance calculation, it does not change our key conclusions. Fig. 6.6a and Fig. 6.6b characterizes the importance of actors and driving scenes across the nuScenes datasets using CDF plots. From Fig. 6.6a, we find that importance is low ($<$0.25) for approximately 85% of actors and only 0.7% of the actors have importance $>$0.9. Similarly, the overall importance of a driving scene is low for 65% of driving scenes. However, only 1.4% of the driving scenes have overall importance of $>$0.9. A driving scene can be important even when importance of individual actor is low because multiple actors together can significantly reduce the driving flexibility of the Ego actor.

6.5 EXAMPLE DRIVING SCENE

Fig. 6.7 shows the importance of a driving scene from nuScnenes dataset. The level of importance for an actor is depicted by a heat map; red being the most important and white being least important. Visually, it make sense for the actors that are nearby to have higher importance value. However, note that not all actors that are closer to the Ego actor (circular object) are equally important. For example, an actor that is moving in the opposite direction despite being closer is less important than the actor that is father away but merging on the same lane.

Figure 6.7: Case study driving scene from nuScenes dataset

6.6 MITIGATING SAFETY HAZARDS

The proposed importance metric allows us to mitigate safety hazards proactively. We develop a safety engine that (i) monitors the importance of each actor at runtime and (ii) disengages the Ego agent sufficiently in advance to mitigate the potential safety hazard safely. The safety engine uses a threshold-based intervention strategy in which the Ego-agent is disengaged when the sum of predicted importance of all actors in a driving scenario increases significantly. In our experiments, the threshold is set to 0.8. We evaluated the safety engine on five driving scenarios shown in Fig. 6.8. Four of these scenarios are from nuScenes dataset and one is a custom cut-in scenario. The custom scenario was created to inject significant uncertainty in the future prediction of one of the actors. We only replicated 4 nuScenes scenarios as replicating each scenario in a simulator involves significant human effort.

The safety engine prevented an accident by disengaging the Ego agent in four out of five driving scenarios. In these scenarios, without the safety engine, the Ego agent's driving decisions lead to accidents. In all cases, the safety engine disengaged the Ego actor at least one second prior to the accident[5]. Our experiments show that the importance metric can be used for mitigating safety hazards proactively. However, a thorough integration with a real vehicle and field evaluations are needed to identify the usability of the proposed metric in the wild.

[5]The time of the accident is identified by running the Ego agent without the safety engine.

6.7 ACCELERATING ASSESSMENT

Importance assessment can further accelerate the fault injection-based assessment techniques such as *Bayesian Fault Injection* (BFI) that was proposed in Chapter 3. BFI gives equal importance to all actors in the scene when estimating the probability of collision under the influence of a fault. However, as discussed above, only few actors ($\sim$ 0.7%) are important while driving. Hence, we integrate the importance assessment technique with BFI to significantly reduce the number of fault injections (i.e., reduce the number of what-if analysis on the model) as shown in Fig. 6.9. Importance-driven BFI accelerates the assessment by 24× when compared to vanilla BFI. Note that this is an additional 24× acceleration over 3690× acceleration that BFI provides compared to existing fault injection techniques.

Figure 6.8: Test driving scenarios for evaluating the safety engine.

6.8 FUTURE WORK

In future, we plan to use this metric in the following ways:

(a) Safety hazard mitigation: We will use Partially Observable Markov Decision Process (POMDP) instead of threshold-based heuristics to develop a safety engine that uses importance metric to identify the most important actors and mitigates any risk posed by those actors.

(b) *Importance-aware planning*: We plan to develop methods that will importance directly with the planner to plan a driving trajectory with the highest driving flexibility while maximizing reward (i.e., reaching the destination).

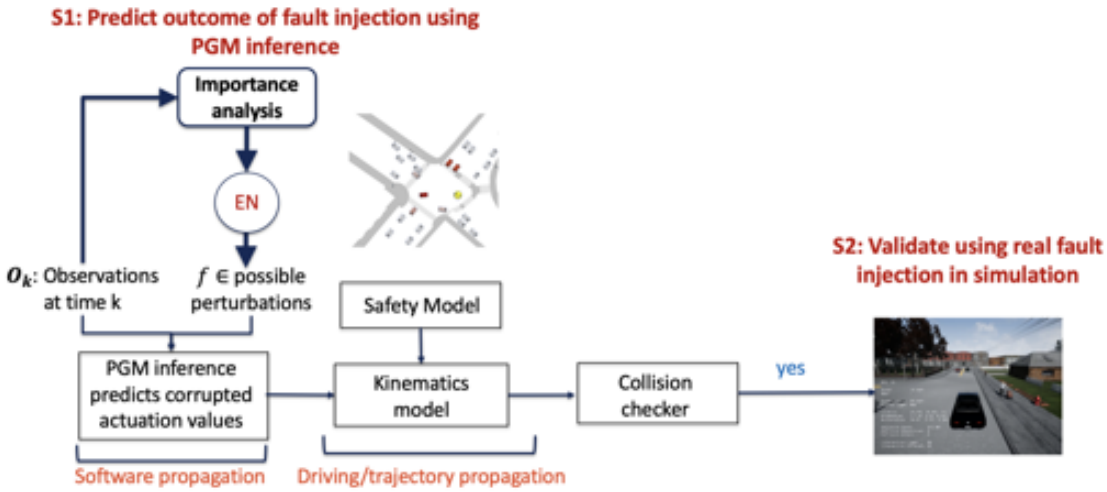


Figure 6.9: Accelerating fault injection-based fault assessment using importance analysis.

(c) *Importance-aware compute resource utilization*: Since at any given point in time, only some actors are important, we plan to use our metric for deciding on how to allocate compute resources so as to generate robust predictions and plans.

We also plan to demonstrate our metric on more realistic agents whose planners not only consider obstacles but rules of the road and maps for navigation.

6.9 CONCLUSION

Driving in a dynamic environment that consists of other actors is inherently a risky task as each actor influences the driving decision and may significantly limit the number of choices in terms of navigation and safety plan. However, not all objects pose a similar risk. Depending on the object's type, trajectory, position, and the associated uncertainty with these quantities; some objects pose a much higher risk than others. In this chapter, we propose a metric that captures the *importance* of each actor in the world with respect to their ability to create safety hazard. In particular, the *importance* metric characterizes the decrease in Ego actor's driving flexibility with respect to a given NPC or a driving scenario. The more constrained the Ego actor higher is the chance of a safety hazard, and therefore, higher is the risk.

CHAPTER 7: HPC: FIELD MEASUREMENTS ON NETWORK

While it is widely acknowledged that network congestion in High Performance Computing (HPC) systems can signifi- cantly degrade application performance, there has been little to no quantification of congestion on credit-based intercon- nect networks. We present a methodology for detecting, ex- tracting, and characterizing regions of congestion in networks. We have implemented the methodology in a deployable tool, Monet, which can provide such analysis and feedback at run- time. Using Monet, we characterize and diagnose congestion in the world's largest 3D torus network of Blue Waters, a 13.3- petaflop supercomputer at the National Center for Supercomputing Applications. Our study deepens the understanding of production congestion at a scale that has never been evaluated before.

7.1 INTRODUCTION

High-speed interconnect networks (HSN), e.g., Infiniband [240] and Cray Aries [241]), which uses credit-based flow control algorithms [242, 243], are increasingly being used in high-performance datacenters (HPC [244] and clouds [245–248]) to support the low-latency communication primitives required by extreme-scale applications (e.g., scientific and deep-learning applications). Despite the network support for low-latency communication primitives and advanced congestion mitigation and protection mechanisms, significant performance variation has been observed in production systems running real-world workloads. While it is widely acknowledged that network congestion can significantly degrade application performance [249–253], there has been little to no quantification of congestion on such interconnect networks to understand, diagnose and mitigate congestion problems at the application or system-level. In particular, tools and techniques to perform runtime measurement and characterization and provide runtime feedback to system software (e.g., schedulers) or users (e.g., application developers or system managers) are generally not available on production systems. This would require continuous system-wide, data collection on the state of network performance and associated complex analysis which may be difficult to perform at runtime.

The core contributions of this chapter are (a) a methodology, including algorithms, for quantitative characterization of congestion of high-speed interconnect networks; (b) introduction of a deployable toolset, *Monet* [254], that employs our congestion characterization methodology; and (c) use of the the methodology for characterization of conges-

tion using 5 months of operational data from a 3D torus-based interconnect network of Blue Waters [49–51], a 13.3-petaflop Cray supercomputer at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. The novelty of our approach is its ability to use *percent time stalled* (P_{Ts})[1] metric to detect and quantitatively characterize congestion hotspots, also referred to as *congestion regions (CRs)*, which are group of links with similar levels of congestion.

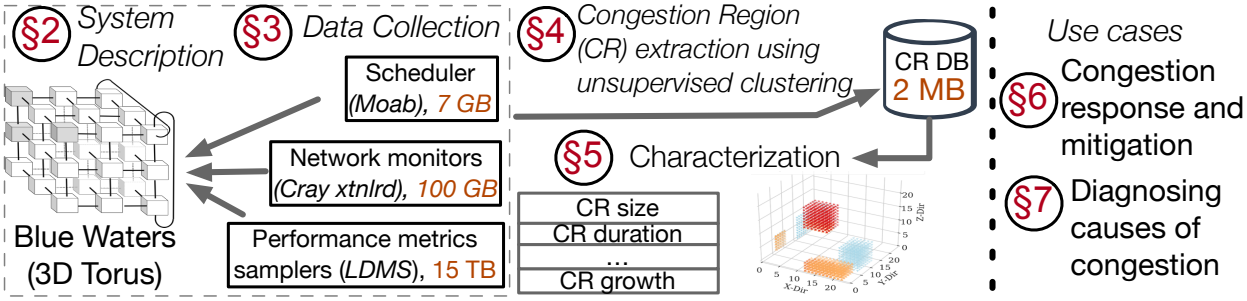


Figure 7.1: Characterization and diagnosis workflow for interconnection-networks.

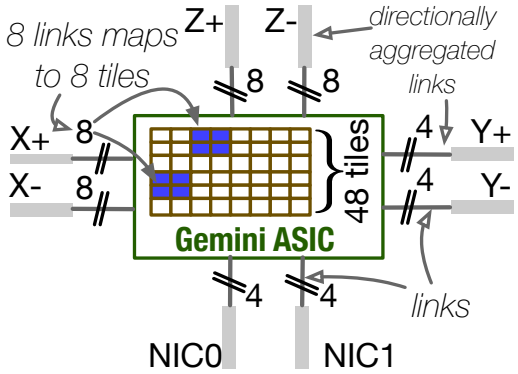


Figure 7.2: Cray Gemini 48-port switch.

The *Monet* tool has been experimentally used on NCSA's Blue Waters. Blue Waters uses a Cray Gemini [255] 3D torus interconnect, the largest known 3D torus in existence, that connects 27,648 compute nodes, henceforth referred to as *nodes*. The proposed tool is not specific to Cray Gemini and Blue Waters; it can be deployed on other k-dimensional mesh or toroidal networks, such as TPU clouds [256], Fujitsu TOFU network-based [257, 258] K supercomputer [259] and upcoming post-K supercomputer [260][2]. The key components of our methodology and the *Monet* toolset are as follows:

Data collection tools: On Blue Waters, we use vendor-provided tools (e.g., `gpcdr` [261]),

[1] P_{Ts}, defined formally in Section §7.2, approximately represents the intensity of congestion on a link, quantified between 0% and 100%.

[2] The first post-K supercomputer is scheduled to be deployed in 2021.

along with the Lightweight Distributed Metric Service (LDMS) monitoring framework [262]. Together these tools collect data on (a) the network (e.g., transferred/received bytes, congestion metrics, and link failure events); (b) the file system traffic (e.g., read/write bytes); and (c) the applications (e.g., start/end time). We released raw network data obtained from Blue Waters [263] as well as the associated code for generating *CRs* as artifacts [254]. To the best of our knowledge, this is the first such large-scale network data release for an HPC high-speed interconnect network that uses credit-based flow control.

A network hotspot extraction and characterization tool, which extracts *CRs* at runtime; it does so by using an unsupervised region-growth clustering algorithm. The clustering method requires specification of congestion metrics (e.g., percent time stalled (P_{Ts}) or stall-to-flit ratios) and a network topology graph to extract regions of congestion that can be used for runtime or long-term network congestion characterization.

A diagnosis tool, which determines the cause of congestion (e.g., link failures or excessive file system traffic from applications) by combining system and application execution information with the CR characterizations. This tool leverages outlier-detection algorithms combined with domain-driven knowledge to flag anomalies in the data that can be correlated with the occurrence of CRs.

To produce the findings discussed in this chapter, we used 5 months of operational data on Blue Waters representing more than 815,006 unique application runs that injected more than 70 PB of data into the network. Our key findings are as follows:

- *While it is rare for the system to be globally congested, there is a continuous presence of highly congested regions (CRs) in the network, and they are severe enough to affect application performance*. Measurements show that (a) for more than 56% of system uptime, there exists at least one highly congested CR (i.e., a CR with a $P_{Ts} > 25\%$), and that these CRs have a median size of 32 links and a maximum size of 2,324 links (5.6% of total links); and (b) highly congested regions may persist for more than 23 hours, with a median duration time of 9 hours[3]. With respect to impact on applications, we observed 1000-node production runs of the NAMD [264] application [4] slowing down by as much as $1.89\times$ in the presence of high congestion compared to median runtime of 282 minutes.
- *Once congestion occurs in the network, it is likely to persist rather than decrease, leading to long-lived congestion in the network*. Measurements show that once the network has entered a state of high congestion ($P_{Ts} > 25\%$), it will persist in high congestion state with a probability of 0.87 in the next measurement window.
- *Quick propagation of congestion can be caused by network component failures.* Network com-

[3]Note that Blue Waters allows applications to run for a maximum of 48 hours.
[4]NAMD is the top application running on Blue Waters consuming 18% of total node-hours [265].

ponent failures (e.g., network router failures) that occur in the vicinity of a large-scale application can lead to high network congestion within minutes of the failure event. Measurements show that 88% of directional link failures [5] caused the formation of CRs with an average $P_{Ts} \geq 15\%$.

- *Default congestion mitigation mechanisms have limited efficacy.* Our measurements show that (a) 29.8% of the 261 triggers of vendor-provided congestion mitigation mechanisms failed to alleviate long-lasting congestion (i.e., congestion driven by continuous oversubscription, as opposed to isolated traffic bursts), as they did not address the root causes of congestion; and (b) vendor-provided mitigation mechanisms were triggered in 8% (261) of the 3,390 high-congestion events identified by our framework. Of these 3,390 events, 25% lasted for more than 30 minutes. This analysis suggests that augmentation of the vendor-supplied solution could be an effective way to improve overall congestion management.

In this chapter, we highlight the utility of congestion regions in the following ways:

- We showcase the effectiveness of *CRs* in detecting long-lived congestion. Based on this characterization, we propose that *CR* detection could be used to trigger congestion mitigation responses that could augment the current vendor-provided mechanisms.
- We illustrate how *CRs*, in conjunction with network traffic assessment, enable congestion diagnosis. Our diagnosis tool attributes congestion cause to one of the following: (a) system issues (such as launch/exit of application), (b) failure issues (such as network link failures), and (c) intra-application issues (such as changes in communication patterns within an application). Such a diagnosis allows system managers to take cause-specific mitigating actions.

7.2 CRAY GEMINI NETWORK AND BLUE WATERS

A variety of network technologies and topologies have been utilized in HPC systems (e.g., [255, 241, 266, 267, 257, 268–270]). Depending on the technology, routing within these networks may be statically defined for the duration of a system boot cycle, or may dynamically change because of congestion and/or failure conditions. The focus of this chapter is on NCSA's Cray XE/XK Blue Waters [50] system, which is composed of 27,648 nodes and has a large-scale (13,824 x 48 port switches) Gemini [255] 3D torus (dimension 24x24x24) interconnect. It is a good platform for development and validation of congestion analysis/ characterization methods as:

[5] see Section §7.5.4 for the definition of directional link.

- It uses directional-order routing, which is predominantly static[6]. From a traffic and congestion characterization perspective, statically routed environments are easier to validate than dynamic and adaptive networks.
- Blue Waters is the best case torus to study since it uses topology-aware scheduling (TAS) [271, 272], discussed later in this section, which has eliminated many congestion issues compared to random scheduling.
- Blue Waters performs continuous system-wide collection and storage of network performance counters.

7.2.1 Gemini Network

In Cray XE/XK systems, four *nodes* are packaged on a *blade*. Each *blade* is equipped with a mezzanine card. This card contains a pair of *Gemini [255] ASICs*, which serve as network switches. The Gemini switch design is shown in Fig. 7.2. Each Gemini ASIC consists of 48 *tiles*, each of which provide a duplex link. The switches are connected with one another in 6 directions, X+/-, Y+/- and Z+/-, via multiple links that form a 3D torus. The number of links in a direction, depends on the direction as shown in the figure; there are 8 each in X+/- and, Z+/- and 4 each in Y+/-. It is convenient to consider all links in a given direction as a *directionally aggregated link*, which we will henceforth call a *link*. The available bandwidth on a particular link is dependent on the link type, i.e., whether the link connects compute cabinets or *blades*, in addition to the number of tiles in the link [273]. X, Y links have aggregate bandwidths of 9.4 GB/s and 4.7 GB/s, respectively, whereas Z links are predominantly 15 GB/s, with 1/8 of them at 9.4 GB/s. Traffic routing in the Gemini network is largely static and changes only when failures occur that need to be routed around. Traffic is directionally routed in the X, Y, and Z dimensions, with the shortest path in terms of hops in + or - chosen for each direction. A deterministic rule handles tie-breaking.

To avoid data loss in the network [7], the Gemini HSN uses a credit-based flow control mechanism [242], and routing is done on a per-packet basis. In credit-based flow control networks, a source is allowed to send a quantum of data, e.g., a flit, to a next hop destination only if it has a sufficient number of credits. If the source does not have sufficient credits, it must stall (wait) until enough credits are available. Stalls can occur in two dif-

[6]When network-link failures occur, network routes are recomputed; that changes the route while the system is up.

[7]The probability of loss of a quantum of data in credit-flow networks is negligible and mostly occurs due to network-related failures.

ferent places: within the switch (resulting in a *inq stall*) or between switches (resulting in an *credit stall*).

Definition 7.1. : A Credit stall is the wait time associated with sending of a flit from an output buffer of one switch to an input buffer of another across a link.

Definition 7.2. : An Inq stall is the wait time associated with sending of a flit from the output buffer of one switch port to an input buffer of another between tiles within the same network switch ASIC.

Congestion in a Gemini-based network can be characterized using both *credit* and *inq* stall metrics. Specifically, we consider the *Percent Time Stalled* as a metric for quantifying congestion, which we generically refer to as the *stall value*.

Definition 7.3. : Percent Time Stalled (P_{Ts}) is the average time spent stalled (T_{is}) over all *tiles* of a directional network link or individual intra-Gemini switch link over the same time interval (T_i): $P_{Ts} = 100 * T_{is}/T_i$.

Depending on the network topology and routing rules, (a) an application's traffic can pass through switches not directly associated with its allocated nodes, and multiple applications can be in competition for bandwidth on the same network links; (b) stalls on a link can lead to back pressure on prior switches in communication routes, causing congestion to spread; and (c) the initial manifestation location of congestion cannot be directly associated with the cause of congestion. Differences in available bandwidth along directions, combined with the directional-order routing, can also cause back pressure, leading to varying levels of congestion along the three directions.

7.2.2 Congestion Mitigation

Run-time evaluations that identify localized areas of congestion and assess congestion duration can be used to trigger *Congestion Effect Mitigating Responses (CEMRs)*, such as resource scheduling, placement decisions, and dynamic application reconfiguration. While we have defined a CEMR as a response that can be used to minimize the negative effects of network congestion, Cray provides a software mechanism [274] to directly alleviate the congestion itself. When a variety of network components (e.g., tiles, NICs) exceeds a high-watermark threshold with respect to the ratio of stalls to forwarded flits, the software instigates a *Congestion Protection Event* (CPE), which is a *throttling* of injection of traffic from all NICs. The CPE mechanism limits the aggregate traffic injection bandwidth

over *all* compute nodes to less than what can be ejected to a *single* node. While this ensures that the congestion is at least temporarily alleviated, the network as a whole is drastically under-subscribed for the duration of the *throttling*. As a result, the performance of all applications running on the system can be significantly impacted. *Throttling* remains active until associated monitored values and ratios drop below their low-watermark thresholds. Applications with sustained high traffic injection rates may induce many CPEs, leading to significant time spent in globally throttling. Bursts of high traffic injection rates may thus trigger CPEs, due to localized congestion, that could have been alleviated without the global negative impact of *throttling*. There is an option to enable the software to terminate the application that it determines is the top congestion candidate, though this feature is not enabled on the Blue Waters system. The option to terminate application in a production environment is not acceptable to most developers and system managers as it will lead to loss of computational node-hours used by the application after the last checkpoint.

While some of this congestion may be alleviated by CEMRs such as feedback of congestion information to applications to trigger rebalancing [275] or to scheduling/resource managers to preferentially allocate nodes (e.g., via mechanisms such as slurm's [276] node weight), some may be unavoidable since all networks have finite bandwidth.

On Blue Waters a topology-aware scheduling (TAS) [271, 272] scheme is used to decrease the possibility of application communication interference by assigning, by default [277], node allocations that are constrained within small-convex prisms with respect to the HSN topology. Jobs that exceed half a torus will still route outside the allocation and possibly interfere with other jobs and vice versa; a non-default option can be used to avoid placement next to such jobs. The I/O routers represent fixed, and roughly evenly distributed, proportional portions of the storage subsystem. Since the storage subsystem components, including I/O routers, are allocated (for writes) in a round robin (by request order) manner independent of TAS allocations, storage I/O communications will generally use network links both within and outside the geometry of the application's allocation and can also be a cause of interference between applications.

7.3 DATA SOURCES AND DATA COLLECTION TOOLS

This section describes the datasets and tools used to collect data at scale to enable both runtime and long-term characterization of network congestion. We leverage vendor-provided and specialized tools to enable collection and real-time streaming of data to

a remote compute node for analysis and characterization. Data provided or exposed on all Cray Gemini systems includes: OS and network performance counter data, network resilience-related logs, and workload placement and status logs. In this study, we used five months (Jan 01 to May 31, 2017) of production network performance-related data (15 TB), network resilience-related logs (100 GB), and application placement logs (7 GB). Note that the methodologies addressed in this work rely only on the availability of the data, independent of the specific tools used to collect the data.

Network Performance Counters: Network performance-related information on links is exposed via Cray's `gpcdr` [261] kernel module. Lustre file system and RDMA traffic information is exposed on the nodes via `/proc/fs` and `/proc/kgnilnd`. It is neither collected nor made available for analysis via vendor-provided collection mechanisms. On Blue Waters, these data are collected and transported off the system for storage and analysis via the Lightweight Distributed Metric Service (LDMS) monitoring framework [262]. In this work, we use the following information: directionally aggregated network traffic (bytes and packets) and length of stalls due to credit depletion; Lustre file system read and write bytes; and RDMA bytes transmitted and received. LDMS samplers collect those data at 60-second intervals and calculate derived metrics, such as the percent of time spent in stalls (P_{Ts}) and percent of total bandwidth used over the last interval. LDMS daemons synchronize their sampling to within a few *ms* (neglecting clock skew) in order to provide coherent *snapshots* of network state across the whole system.

Network Monitoring Logs: Network failures and congestion levels are monitored and mitigated by Cray's *xtnlrd* software. This software further logs certain network events in a well-known format in the *netwatch* log file. Significant example log lines are provided in Cray documents [278, 274]. Regular expression matching for these lines is implemented in LogDiver [279], a log-processing tool, which we use to extract the occurrences, times, and locations of link failures and CPEs.

Workload Data: Blue Waters utilizes the Moab scheduler, from which application queue time, start time, end time, exit status, and allocation of nodes can be obtained. The workload dataset contains information about 815,006 application runs that were executed during our study period.

Note that we will only be releasing network data. Worload data and network monitoring logs will not be released due to privacy and other concerns.

7.4 *CR* EXTRACTION AND CHARACTERIZATION TOOL

This section first describes our motivation for choosing congestion regions (CRs) as a driver for characterizing network congestion, and then describes our methodology (implemented as the *Monet* tool) for extracting *CRs* over each data collection interval and the classification of those CRs based on severity.

7.4.1 Why Congestion Regions?

We seek to motivate our choice to characterize congestion regions (*CRs*) and the need for estimates for severity in terms of the stall values. We first show that the characterization of hotspot links individually do not reveal the spatial and growth characteristics which is needed for diagnosis. Then, we show how characterizing *CRs* is meaningful.

Characterizing hotspot links individually do not reveal regions of congestion. Figure Fig. 7.3 characterizes the median, 99%ile and 99.9%ile duration of the hotspot links by generating the distribution of the duration for which a link persists to be in congestion at $P_{Ts} \geq P_{Ts}$Threshold value. For example, 99.9%ile duration for hotspot links with $P_{Ts} \geq 30$ is 400 minutes (6.67 hours). The measurements show that the median duration of hotspot link at different P_{Ts} thresholds is constantly at ~ 0, however, 99.9%ile duration of hotspot links linearly decreases with increasing P_{Ts} threshold value. Although such characterizations are useful to understand congestion at link-level, they hide the spatial characteristics of congestion such as the existence of multiple pockets of congestion and their spread and growth over time. The lack of such information makes it difficult to understand congestion characteristics and their root cause.

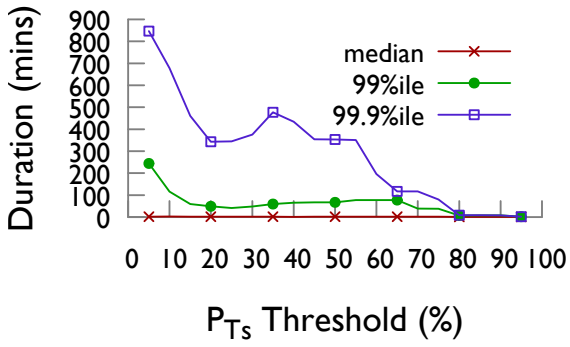
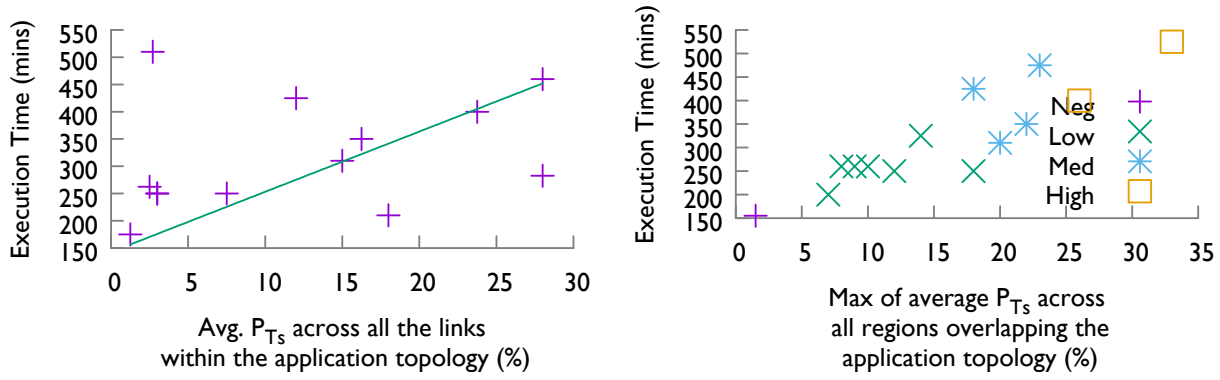


Figure 7.3: Duration of congestion on links at different P_{Ts} thresholds



(a) Not using *CRs*

(b) Using *CRs*

Figure 7.4: Correlating congestion with NAMD application runtime

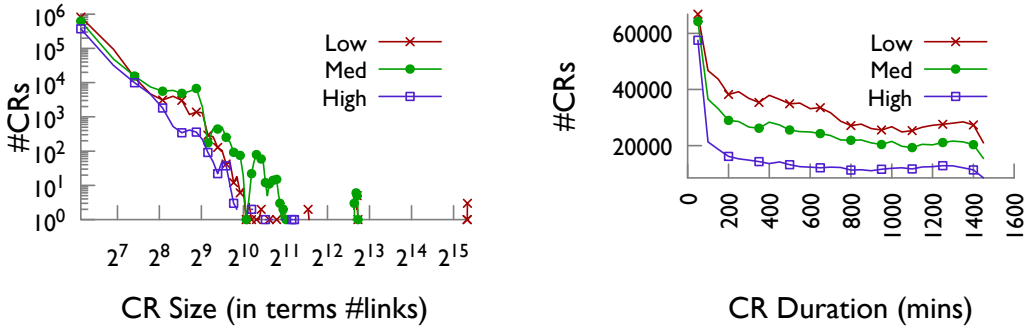
***CRs* captures relationship between congestion-level and application slowdown efficiently.** In order to determine possible severity values and show effectiveness of *CRs* in determining application slowdown, we extracted from the production Blue Waters dataset a set of NAMD [264][8] runs each of which ran on 1000 nodes with the same input parameters. We chose NAMD because it consumes approximately 18% of total node-hours available on Blue Waters[9]. Fig. 7.4a shows the execution time of each individual run with respect to the *average* P_{Ts} over all links within the allocated application topology. (Here we leverage TAS to determine severity value estimates based on the values within the allocation; that is not a condition for the rest of this work.) Fig. 7.4a shows that execution time is perhaps only loosely related to the average P_{Ts}; with correlation of 0.33 . In contrast, Fig. 7.4b shows the relationship of the application execution time with the *maximum* average P_{Ts} over all *CRs* (defined in §7.4.2) within the allocated topology; with correlation of 0.89. In this case, execution time increases with increasing maximum of average P_{Ts} over all regions. We found this relationship to hold for other scientific applications. This is a motivating factor for the extraction of such *congestion regions* (*CRs*) as indicators of 'hot-spots' in the network. We describe the methodology for *CR* extraction in the next section.

In addition, we selected approximate ranges of P_{Ts} values, corresponding to increasing

[8]NAMD has two different implementations: (a) uGNI shared memory parallel (SMP)-based, and (b) MPI-based. In this work, unstated NAMD refers to uGNI SMP-based implementation. uGNI is user level Generic Network Interface [280].

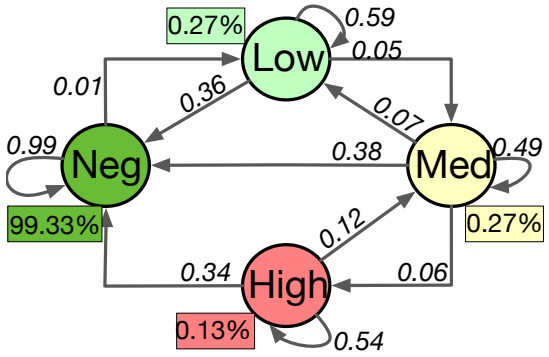
[9]This was best effort extraction and the NAMD application runs may not be exactly executing the same binary or processing the same data, as user may have recompiled the code with a different library or used the same name for dataset while changing the data. There is limited information to extract suitable comparable runs from historical data that are also subject to allocation and performance variation.

run times, to use as estimates for the severity levels as these can be easily calculated, understood and compared. These levels are indicated as symbols in the figure. Explicitly, we assign 0-5% average P_{Ts} in a *CR* as *Negligible* or *'Neg'*, 5-15% as *'Low'*, 15-25% as *'Medium'*, and $> 25\%$ as *'High'*. These are meant to be qualitative assignments and not to be rigorously associated with a definitive performance variation for all applications in all cases, as the network communication patterns and traffic volumes vary among HPC applications. We will use these ranges in characterizations in the rest of this work. More accurate determinations of impact could be used in place of these in the future, without changing the validity of the *CR* extraction technique.



(a) Distribution of CR sizes.

(b) Distribution of CRs.



(c) *CR* congestion evolution captures transition probabilities. Percentage in boxes indicates percentage of total link-hours spent in that *state*.

Figure 7.5: *CR* size, duration, evolution characterization. # of *CRs* across *'Low'*, *'Medium'*, and *'High'* are 9.4e05, 7.3e05, and 4.2e05 respectively.

7.4.2 Extracting Congestion Regions

We have developed an unsupervised clustering approach for extracting and localizing regions of congestion in the network by segmenting the network into groups of links with

similar congestion values. The clustering approach requires the following parameters: (a) network graph (G), (b) congestion measures (v_s for each vertex v in G), (c) neighborhood distance metric (d_δ), and (d) stall similarity metric (d_λ). The network is represented as a graph G. Each link in the network is represented as a vertex v in G, and two vertices are connected if the corresponding network links are both connected to the same switch (i.e., the switch is an edge in the graphs). For each vertex v, the congestion measures(s) are denoted by the vector v_s, which is composed of credit stalls and inq stalls, which we use independently. Distance metrics d_δ and d_λ are also required, the former for calculating distances between two vertices and the latter for calculating differences among the stalls v_s. We assign each vertex the coordinate halfway between the logical coordinates of the two switches to which that vertex is immediately connected, and we set d_δ to be the L1 norm between the coordinates. Since the Blue Waters data consists of directionally aggregated information as opposed to counters on a per-tile-link (or buffer) basis, then, in our case, d_λ is simply the absolute difference between the two credit-stall or the two inq-stall values of the links, depending on what kinds of regions are being segmented. We consider credit and inq stalls separately to extract *CRs*, as the relationship between the two types of stalls is not immediately apparent from the measurements, and thus require two segmentation passes. Next, we outline the segmentation algorithm.

Segmentation Algorithm The segmentation algorithm has four stages which are executed in order, as follows.

- Nearby links with similar stall values are grouped together. Specifically, they are grouped into the equivalence classes of the reflexive and transitive closure of the relation $\sim_r$ defined by $x \sim_r y \Leftrightarrow d_\delta(x, y) \leq \delta \wedge d_\lambda(x_s - y_s) \leq \theta_p$, where x, y are vertices in G, and δ, θ_p are thresholds for distance between vertices and stall values, respectively.
- Nearby regions with similar average stall values, are grouped together through repetition of the previous step, but with regions in place of individual links. Instead of using the link values v_s, we use the average value of v_s over all links in the region, and instead of using θ_p, we use a separate threshold value θ_r.
- CRs that are below the size threshold σ are merged into the nearest region within the distance threshold δ.
- Remaining CRs with $< \sigma$ links are discarded, so that regions that are too small to be significant are eliminated.

The optimum values for the parameters used in segmentation algorithms, except for δ, were estimated empirically by knee-curve [281] method, based on the number of regions

produced. Using that method, the obtained parameter values [10] are: (a) $\theta_p = 4$, (b) $\theta_r = 4$, and (c) $\sigma = 20$. In [281], the authors conclude that the optimum sliding window time is the knee of the curve drawn between the sliding window time and the number of clusters obtained using a clustering algorithm. This decreases truncation errors (in which a cluster is split into multiple clusters because of a small sliding window time) and collision errors (in which two events not related to each other merge into a single cluster because of a large sliding window time). We fixed δ to be 2 in order to consider only links that are two hops away, to capture the local nature of congestion [282]. It should be noted that the region clustering algorithm may discard small isolated regions (size $\leq \sigma$) of high congestion. If such *CRs* do cause high interference, they will grow over time and eventually be captured.

Our algorithm works under several assumptions: (a) congestion spreads locally, and (b) within a CR, the stall values of the links do not vary significantly. These assumptions are reasonable for k-dimensional toroids that use directional-order routing algorithm. The methodology used to derive *CRs* is not dependent on the resource allocation policy (such as TAS). The proposed extraction and its use for characterization is particularly suitable for analysis of network topologies that use directional- or dimensional-order routing. In principle, the algorithm can be applied to other topologies (such as mesh and high-order torus networks) with other metrics (such as stall-to-flit ratio). Furthermore, the region extraction algorithm does not force any shape constraints; thus *CRs* can be of any arbitrary shape requiring us to store each vertex associated with the *CR*. In this work, we have configured the tool to store and display bounding boxes over CRs, as doing so vastly reduces the storage requirements (from TBs of raw data to 4 MB in this case), provides a succinct summary of the network congestion state, and eases visualization.

We validate the methodology for determining the parameters of the region-based segmentation algorithm and its applicability for *CR* extraction by using a synthetic dataset.

7.4.3 Implementation and Performance

We have implemented the region-extraction algorithm as a modified version of the region growth segmentation algorithm [283] found in the open-source PointCloud Library (PCL) [284] [285]. The tool is capable of performing run-time extraction of *CRs* even for large-scale topologies. Using the Blue Waters dataset, Monet mined *CRs* from each 60-

[10] stall thresholds are scaled by $2.55\times$ to represent the color range (0-255) for visualization purposes

second snapshot of data for 41,472 links in ~ 7 seconds; Monet was running on a single thread of a 2.0 GHz Intel Xeon E5-2683 v3 CPU with 512 GB of RAM. Thus, on Blue Waters *Monet* can be run at run-time, as the collection interval is much greater than CR extraction time. Since Monet operates on the database, it works the same way whether the data are being streamed into the database or it is operating on historical data.

7.5 CHARACTERIZATION RESULTS

In this section, we present results of the application of our analysis methodology to five months of data from a large-scale production HPC system (Blue Waters) to provide characterizations of *CRs*. Readers interested in understanding traffic characteristics at the link and datacenter-level may refer to a related work [286].

7.5.1 Congestion Region Characterization

Here we assess and characterize the congestion severity.

CR-level Size and Severity Characterizations: Fig. 7.5a shows a histogram[11] of *CR* sizes in terms of the number of links for each congested *state* (i.e., not including *'Neg'*). Fig. 7.5b show a histogram of the durations of *CRs* across *'Low'*, *'Medium'* and *'High'* congestion levels. These measurements show that unchecked congestion in credit-based interconnects leads to:

- High growth and spread of congestion leading to large *CRs*. The max size of *CRs* in terms of number of links was found to be 41,168 (99.99% of total links), 6,904 (16.6% of total links), and 2,324 (5.6% of total links) across *'Low'*, *'Medium'* and *'High'* congestion levels respectively, whereas the 99th percentile of the[12] *CR* size was found to be 299, 448, and 214 respectively.
- Localized congestion hotspots, i.e., pockets of congestion. *CRs* rarely spread to cover all of the network. The number of *CRs* decreases (see Fig. 7.5a) with increasing size across all severity *states* except for *'Low'* for which we observe increase at the tail. For example, there are $\sim 16,000$ CRs in the *'High'* which comprise 128 links but only ~ 141 CRs of size ~ 600.
- Long-lived congestion. The *CR* count decreases with increasing duration, however there are many long-lived *CRs*. The 50%ile, 99%ile and max duration of *CRs* across all

[11] plotted as lines and every tenth point marked on the line using a shape for clarity.
[12] We will use %ile to denote percentile in the rest of the chapter.

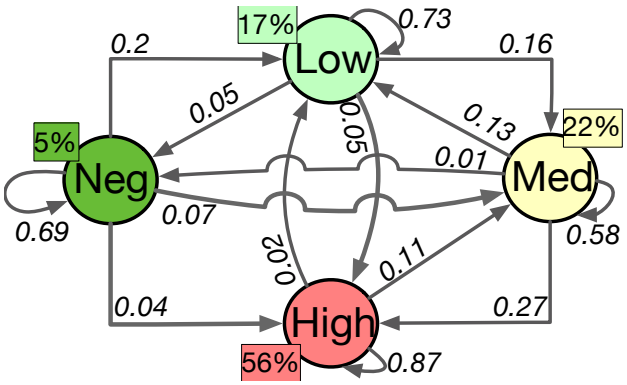


Figure 7.6: *Network* congestion evolution captures transition probabilities from one severity *state* to another. Percentage numbers in boxes indicates percentage of total system wall clock time spent in that *state*.

states were found to be 579 minutes (9.7 hours), 1421 minutes (23.6 hours), and 1439 minutes (24 hours) respectively, whereas the 50%ile, 99%ile and max P_{Ts} of *CRs* was found to be 14%, 46%, and 92%, respectively. *CR* duration did not change significantly across *'Low'*, *'Medium'*, and *'High'*.

CR Evolution and State Probability: Fig. 7.5c shows the transition probabilities of the *CR* states. The percentage in the box next to each *state* shows the percentage of total link-hours[13] spent in that *state*. It can be interpreted as the probability that a link will be congested at a severity *state* at a given time. For example, there is a probability of 0.10% that a link will be in the *'High'*. These measurements show that:

- The vast majority of link-hours (99.3% of total link-hours) on Blue Waters are spent in *'Neg'* congestion. Consideration of a grosser congestion metric, such as the average stall time across the entire network, will not reveal the presence of significant *CRs*.
- Once a *CR* of *'Low'*, *'Medium'* or *'High'* congestion is formed, it is likely to persist (with a probability of more than 0.5) rather than decrease or vanish from the network.

7.5.2 Network-level Congestion Evolution and Transition Probabilities

In this section, we assess and characterize the overall network congestion severity state. The overall network congestion severity *state* is the *state* into which the highest CR falls. That assignment is independent of the overall distribution of links in each *state*. Figure Fig. 7.6 shows the probabilities that transitions between network *states* will occur

[13]Link-hours are calculated by $\sum$ (#links in *Region*) $\times$ (measurement time-window) for each *state*.

between one measurement interval and the next. The rectangular boxes in the figure indicate the fraction of time that the network resides in each *state*. These measurements show the following:

- While each *individual* link of the entire network is most often in a *state* of *'Neg'* congestion, there exists at least one *'High'* CR for 56% of the time. However, *'High'* CRs are small; in Section §7.5.1, we found that 99th percentile size of *'High'* is 214 links. Thus, the Blue Waters network *state* is nearly always non-negligible (95%), with the "High" *state* occurring for the majority of the time.
- There is a significant chance that the current network *state* will persist or increase in severity in the next measurement period. For example, there is an 87% chance that it will stay in a *'High' state*.
- A network *state* is more likely to drop to the next lower *state* than to drop to *'Neg'*.
- Together these factors indicate that congestion builds and subsides slowly, suggesting that it is possible to forecast (within bounds) congestion levels. Combined with proactive localized congestion mitigation techniques and CEMRs, such forecasts could significantly improve overall system performance and application throughput.

7.5.3 Application Impact of CR

The potential impact of congestion on applications can be significant, even when the percentage of link-hours spent in non-*'Neg'* congested regions is small. While we cannot quantify congestion's impact on all of the applications running on Blue Waters (as we lack ground truth information on particular application runtimes without congestion), we can quantify the impact of congestion on the following:

- Production runs of the NAMD application [264]. The worst-case NAMD execution runtime was $3.4\times$ slower in the presence of high *CRs* relative to baseline runs (i.e., negligible congestion). The median runtime was found be 282 minutes, and hence worst-case runtime was $1.86\times$ slower than the median runtime. This is discussed in more detail in Section §7.4.1.
- In [286], authors show that benchmark runs of PSDNS [287] and AMR [288] on 256 nodes slowed down by as much as $1.6\times$ even at low-levels of congestion ($5\% < P_{Ts} \leq 15\%$).

To find a upper bound on the number of potentially impacted applications, we consider the applications whose allocations are directly associated with a router in a *CR*. Out of 815,006 total application runs on Blue Waters, over 16.6%, 12.3%, and 6.5% of the unique application runs were impacted by *'Low'*, *'Medium'*, and *'High'* CRs, respectively.

7.5.4 Congestion Scenarios

In this section, we show how *CRs* manifest under different congestion scenarios: (a) system issues (e.g. changes in system load), (b) network-component failures (e.g. link failures), and (c) intra-application contention. in conjunction with applications' placements at runtime on the torus. *CRs* of *'Neg'* congestion are not shown in the figures.

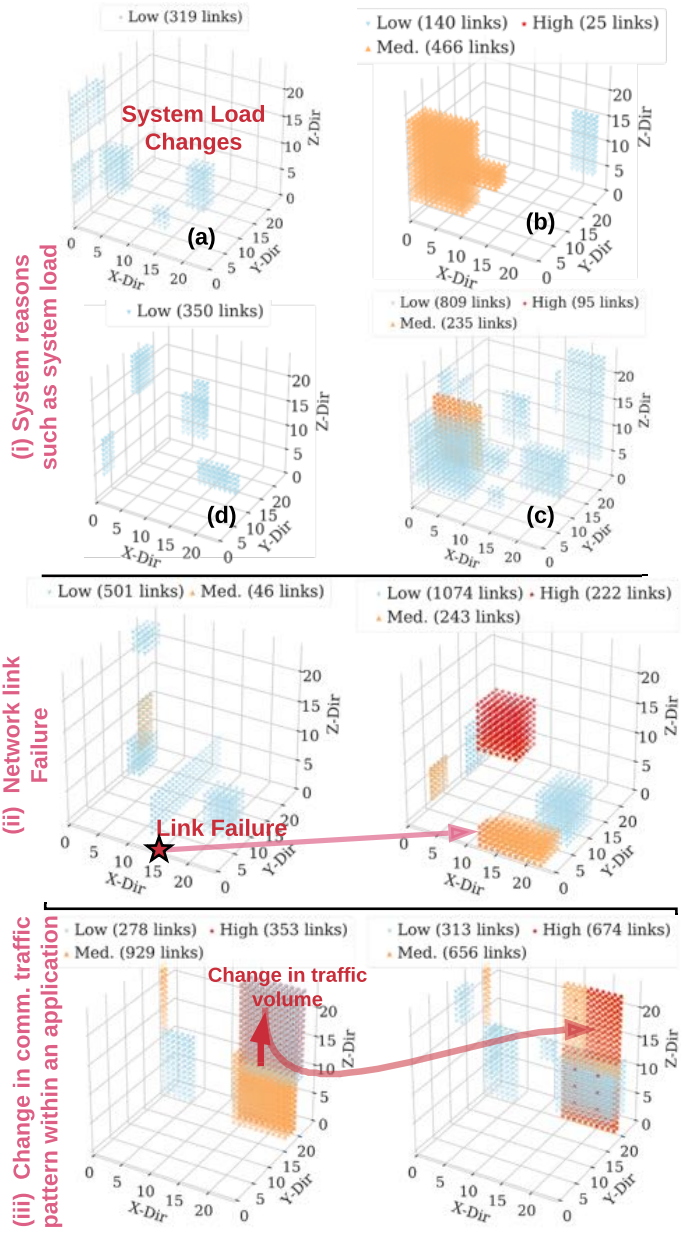


Figure 7.7: Case studies: network congestion is shown due to (i) system issues (such as introduction of new applications), (ii) failures (such as network link failure), and (iii) change in communication pattern within the application.

Congestion due to System Issues: Network congestion may result from contention between different applications for the same network resources. That can occur because of a change in system load (e.g. launches of new applications) or change in application traffic that increases contention on shared links between applications.

Fig. 7.7(i) shows four snapshots, read clockwise, of extracted *CRs*, including size and severity state, for different time intervals during a changing workload. Fig. 7.7(i)(a) shows that *'Low'* (blue) *CRs* when most of the workload consists of multiple instances of MPI-based *NAMD* [264]. The overall network state was thus *'Low'*. The *CRs* remained relatively unchanged for 40 minutes, after which two instances of NAMD completed and *Variant Calling*[289] was launched. Three minutes after the launch, new *CRs* of increased severity occurred (Fig. 7.7(i)(b,c)). The *'High'* (red) [14] and *'Medium'* (orange) severity *CRs* overlapped with the applications.

The increase in the severity of congestion was due to high I/O bandwidth utilization by the *Variant Calling* application. The overall network state remained *'High'* for ~ 143 minutes until the *Variant Calling* application completed. At that time, the congestion subsided, as shown in Fig. 7.7(i)(d).

Congestion Due to Network-component Failures: Network-related failures are frequent [38, 290] and may lead to network congestion, depending on the traffic on the network and the type of failure. In [38], the mean time between failures (MTBF) for directional links in Blue Waters was found to be approximately $2.46e06$ link-hours (or 280 link-years). Given the large number of links (41,472 links) on Blue Waters, the expected mean time between failure of a link across the system is about 59.2 hours; i.e., Blue Waters admins can expect one directional-link failure every 59.2 hours.

Failures of directional links or routers generally lead to occurrences of *'High' CRs*, while isolated failures of a few switch links (which are much more frequent) generally do not lead to occurrences of significant *CRs*. In this work we found that 88% of directional link failures led to congestion; however, isolated failures of switch links did not lead to significant *CRs* (i.e., had *'Neg' CRs*).

Fig. 7.7(ii) shows the impact of a network blade failure that caused the loss of two network routers and about 96 links (x,y,z location of failure at coordinates (12,3,4) and (12,3,3)). Fig. 7.7(ii)(a) shows the congestion *CRs* before the failure incident and Fig. 7.7(ii)(b) shows the *CRs* just after the completion of the network recovery. Immediately after failure, the stalls increased because of the unavailability of links, requiring the packets to

[14] not visible and hidden by other regions.

be buffered on the network nodes. The congestion quickly spread into the geometry of nearby applications in the torus. Failure of a blade increased the overall size (in number of links) of *'Low' CRs* by a factor of 2, and of *'Medium' CRs* by a factor of 4.2, and created previously non existent *'High' CRs* with more than 200 links.

Congestion Due to Intra-Application Issues: Congestion within an application's geometry (intra-application contention) can occur even with TAS. Fig. 7.7(iii) shows congestion *CRs* while the uGNI-based shared memory parallel (SMP) *NAMD* application on more than 2,000 nodes. The application is geometrically mapped on the torus starting at coordinates (15, 18, 0) and ending at coordinates (1, 21, 23) (wrapping around). The congestion *CRs* alternate between the two states shown (state 1 shown in Fig. 7.7(iii)(a), and state 2, shown in Fig. 7.7(iii)(b)) throughout the application run-time because of changes in communication patterns corresponding to the different segments of the NAMD code.

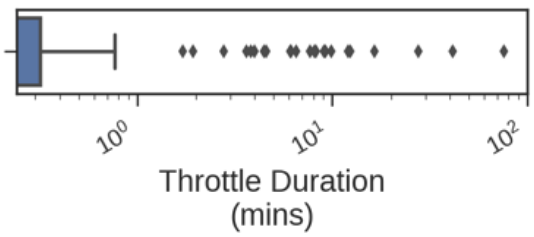
Intra-application contention is less likely to elevate to cause global network issue, unless the links are involved in global (e.g., I/O) routes, or if the resulting congestion is heavy enough to trigger the system-wide mitigation mechanism (see Section §7.2.2).

Importance of diagnosis: In this section, we have identified three high-level causes of congestion, which we categorize as (a) system issues, (b) network-component failures, and (c) intra-application contention. For each cause, system managers could trigger one of the following actions to reduce/manage congestion. In the case of intra-application congestion, an automated MPI rank remapping tool such as TopoMapping [291], could be used to change traffic flow bandwidth on links to reduce congestion on them. In the case of inter-application congestion (caused by system issues or network failures), a node-allocation policy (e.g., TAS) could use knowledge of congested regions to reduce the impact of congestion on applications. Finally, if execution of an application frequently causes inter-application congestion, then the application should be re-engineered to limit chances of congestion.

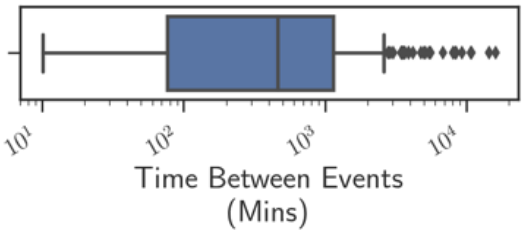
7.6 USING CHARACTERIZATIONS: CONGESTION RESPONSE

In this section, we first discuss efficacy of Cray CPEs and then show how our *CR*-based characterizations can be used to inform effective responses to performance-degrading levels of congestion.

Characterizing Cray *CPEs*: Recall from Section §7.2 that the vendor-provided congestion mitigation mechanism throttles all NIC traffic injection into the network irrespective



(a) Box plot of duration of throttling



(b) Box plot of time between triggers of congestion mitigation events

Figure 7.8: Characterizing Cray Gemini congestion mitigation events.

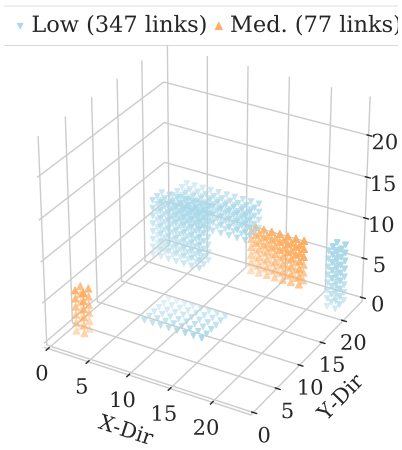
of the location and size of the triggering congestion region. This mitigation mechanism is triggered infrequently by design and hence may miss detections and opportunities to trigger more targeted congestion avoidance mechanisms. On Blue Waters, congestion mitigation events are generally active for small durations (typically less than a minute), however, in extreme cases, we have seen them active for as long as 100 minutes. Each throttling event is logged in *netwatch* log files.

We define a *congestion mitigation event (CME)* as a collection of one or more throttling events that were coalesced together based on a sliding window algorithm [281] with a sliding window of 210 seconds, and we use this to estimate the duration of the vendor-provided congestion mitigation mechanisms. Fig. 7.8a and Fig. 7.8b shows a box plot of duration of and time between *CMEs* respectively. The analysis of *CMEs* shows that :

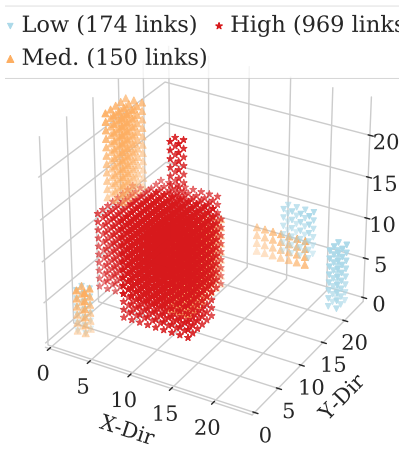
- CMEs were triggered 261 times; 29.8% of which did not alleviate congestion in the system. Fig. 7.9 shows a case where the size and severity of *CRs* increases after a series of throttling events.
- The median time between triggers of CMEs was found to be 7 hours. The distribution of time between events is given in Fig. 7.8b.
- CMEs are generally active for small durations (typically less than a minute), however, in extreme cases, we have seen them active for as long as 100 minutes.
- 8% of the application runs were impacted with over 700 of those utilizing > 100 nodes.

These observations motivate the utility of augmenting the vendor supplied solution of global traffic suppression to manage exceptionally high congestion bursts with our more localized approach of taking action on *CRs* at a higher system-level of granularity to alleviate sources of network congestion.

***CR*-based congestion detection to increase mitigation effectiveness:** *CR* based characterizations can potentially improve congestion mitigation and CEMR effectiveness by

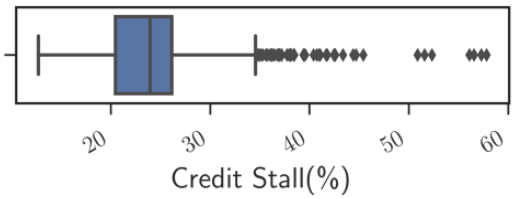


(a) Congestion before triggering of CPE

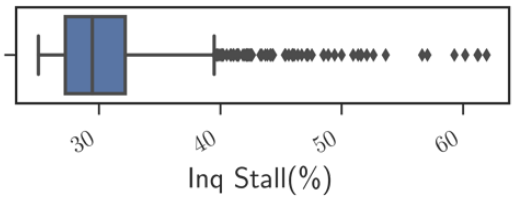


(b) Congestion after 30 minutes of CPE

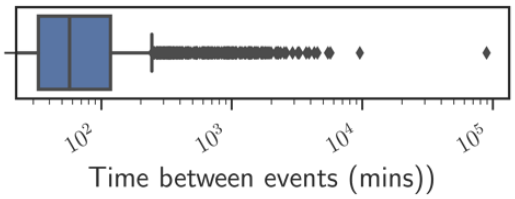
Figure 7.9: A case in which a congestion protection event (CPE) failed to mitigate the congestion



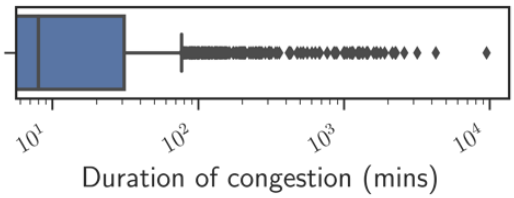
(a) Boxplot of average credit stall across extracted congestion events.



(b) Boxplot of average inq-stall across extracted congestion events.



(c) Boxplot of time between congestion events.



(d) Boxplot of duration of congestion.

Figure 7.10: Characterization of Regions Congestion Events (RCE).

more accurately determining which scenarios should be addressed by which mechanisms and by using the identified *CRs* to trigger localized responses more frequently than Cray *CMEs*. That approach is motivated by our discovery (see Section §7.5.2) that the network is in a *'High'* congestion *state* the majority of the time, primarily because of *CRs* of small size but significant congestion severity.

We define a *Regions Congestion Event (RCE)* as a time-window for which each time in-

stance has at least one region of *'High'* congestion. We calculate it by combining the *CR* evaluations across 5-minute sliding windows. Fig. 7.10 shows boxplots of (a) average credit P_{TS} across all extracted *CRs* during RCEs', (b) average inq P_{TS} across all RCEs', (c) times between RCE, and (d) durations of the RCEs'. These measurements show

- Relative to the vendor-provided congestion mitigation mechanisms, our characterization results in $13\times$ more events (3390 RCEs) upon which we could potentially act.
- Vendor provided congestion mitigation mechanisms trigger on 8% (261 of 3390) of RCEs.
- The average P_{TS} of maximum inq- and credit-stall across all extracted regions present in *RCEs* is quite high, at 33.8% and 27.4%, respectively.
- 25% of 3390 RCEs lasted for more than 30 minutes, and the average duration was found to be approximately an hour.

CRs discovery could also be used for informing congestion aware scheduling decisions. Communication-intensive applications could be preferentially placed to not contend for bandwidth in significantly congested regions or be delayed from launching until congestion has subsided.

7.7 USING CHARACTERIZATIONS: DIAGNOSING CAUSES OF CONGESTION

Section §7.5.4 identifies the root causes of congestion and discusses the the importance of diagnosis. Here we explore that idea to create tools to enable diagnosis at runtime.

7.7.1 Diagnosis Methodology and Tool

We present a methodology that can provide results to help draw a system manager's attention to anomalous scenarios and potential offenders for further analysis. We can combine system information with the *CR*-characterizations to help diagnose causes of significant congestion. Factors include applications that inject more traffic than can be ejected into the targets or than the traversed links can transfer, either via communication patterns (e.g., all-to-all or many-to-one) or I/O traffic, and link failures. These can typically be identified by observation(s) of anomalies in the data.

Mining Candidate Congestion-Causing Factors For each congestion Region, CR_i, identified at time T, we create two tables $\mathscr{A}_{CR_i}(T)$ and $\mathscr{F}_{CR_i}(T)$, as described below.

$\mathscr{A}_{CR_i}(T)$ *table*: Each row in $\mathscr{A}_{CR_i}(T)$ corresponds to an application that is within $N_{hops} \leq$

3 hops away from the bounding box of the congestion region CR_i. $\mathscr{A}_{CR_i}(T)$ contains information about the application and its traffic characteristics across seven *traffic features*: (a) application name, (b) maximum read bytes per minute, (c) maximum write bytes per minute, (d) maximum RDMA read bytes per minute, (e) maximum RDMA write bytes per minute, (f) maximum all-to-all communication traffic bytes per minute, and (g) maximum many-to-one communication traffic bytes per minute, where the maximums are taken over the past 30 minutes, i.e., the most recent 30 measurement windows. The list of applications that are within N_{hops} away from congestion region CR_i are extracted from the *workload data*. The measurements for features (a) to (e) are extracted by querying network performance counter data, whereas we estimate the features (f) and (g) are estimated from Network performance counter data by taking several bisection cuts over the application geometry and comparing node traffic ingestion and ejection bytes among the two partitions of the bisection cut.

$\mathscr{F}_{CR_i}(T)$ *table*: Each row in $\mathscr{F}_{CR_i}(T)$ corresponds to an application that is within $N_{hops} \leq 3$ away from the congestion boundary of CR_i. $\mathscr{F}_{CR_i}(T)$ contains information about failure events across three *failure features*: (a) failure timestamp, (b) failure location (i.e., coordinates in the torus), and (c) failure type (i.e., switch link, network link, and router failures). Lists of failure events that are within N_{hops} away from congestion region CR_i are extracted from *network failure data*.

Identifying Anomalous or Extreme Factors: The next step is to identify extreme application traffic characteristics or network-related failures over the past 30 minutes that have led to the occurrence of CRs. For each traffic feature in $\mathscr{A}_{CR_i}(T)$, we use an outlier detection method to identify the top k applications that are exhibiting anomalous behavior. The method uses the numerical values of the features listed in table $\mathscr{A}_{CR_i}(T)$. Our analysis framework uses a median-based outlier detection algorithm proposed by Donoho [292] for each CR_i. According to [292], the median-based method is more robust than mean-based methods for skewed datasets. Because *CRs* due to network-related failure events [15] are rare relative to congestion caused by other factors, all failure events that occur within N_{hops} of CR_i in the most recent 30 measurement windows are marked as anomalous.

Generating Evidence: The last step is to generate evidence for determining whether anomalous factors identified in the previous step are truly responsible for the observed

[15] In this chapter, we do not consider the effect of lane failures on congestion.

congestion in the *CR*. The evidence is provided in the form of a statistical correlation taken over the most recent 30 measurement time-windows between the moving average stall value of the links and the numerical traffic feature(s) obtained from the data (e.g., RDMA read bytes per minute of the application) associated with the anomalous factor(s). For failure-related anomalous factors, we calculate the correlation taken over the most recent 30 measurement time-windows between the moving average of observed traffic summed across the links that are within N_{hops} away from the failed link(s) and the stall values[16]. A high correlation produces the desired evidence. We order the anomalous factors using the calculated correlation value regardless of the congestion cause. Additionally, we show a plot of stall values and the feature associated with the anomalous factor(s) to help understand the impact of the anomalous factor(s) on congestion.

The steps in this section were only tested on a dataset consisting of the case studies discussed in Section §7.5.4 and §7.7 because of lack of ground truth labels on root causes. Creation of labels on congestion causes requires significant human effort and is prone to errors. However, we have been able to generate labels by using the proposed unsupervised methodology, which provides a good starting point for diagnosis.

7.7.2 Comprehensive Congestion Analysis

In this section, we describe an example use case in which our analysis methodologies were used to detect and diagnose the congestion in a scenario obtained from real data for which the ground truth of the cause was available. The overall steps involved in using our methodologies, included in our *Monet* implementation, for congestion detection and diagnosis are summarized in Fig. 7.11 and described in Section §7.7. Not all of the steps discussed below are currently automated, but we are working on automating an end-to-end pipeline.

Step 1. Extraction of CR. Fig. 7.11(a) shows that our analysis indicated wide spread high-level congestion across the system (see the left graph in Fig. 7.11(a)). An in-depth analysis of the raw data resulted in identification/detection of congestion regions (see the top-right graph in Fig. 7.11(a)).

Step 2. Congestion diagnosis. There are 3 steps associated with diagnosing the cause of the congestion.

Step 2.1. Mining candidate factors. To determine the cause of the congestion, we correlated the CR-data with application-related network traffic (for all applications that over-

[16]Increase in traffic near a failed link leads to congestion as shown in Section §7.5.4.

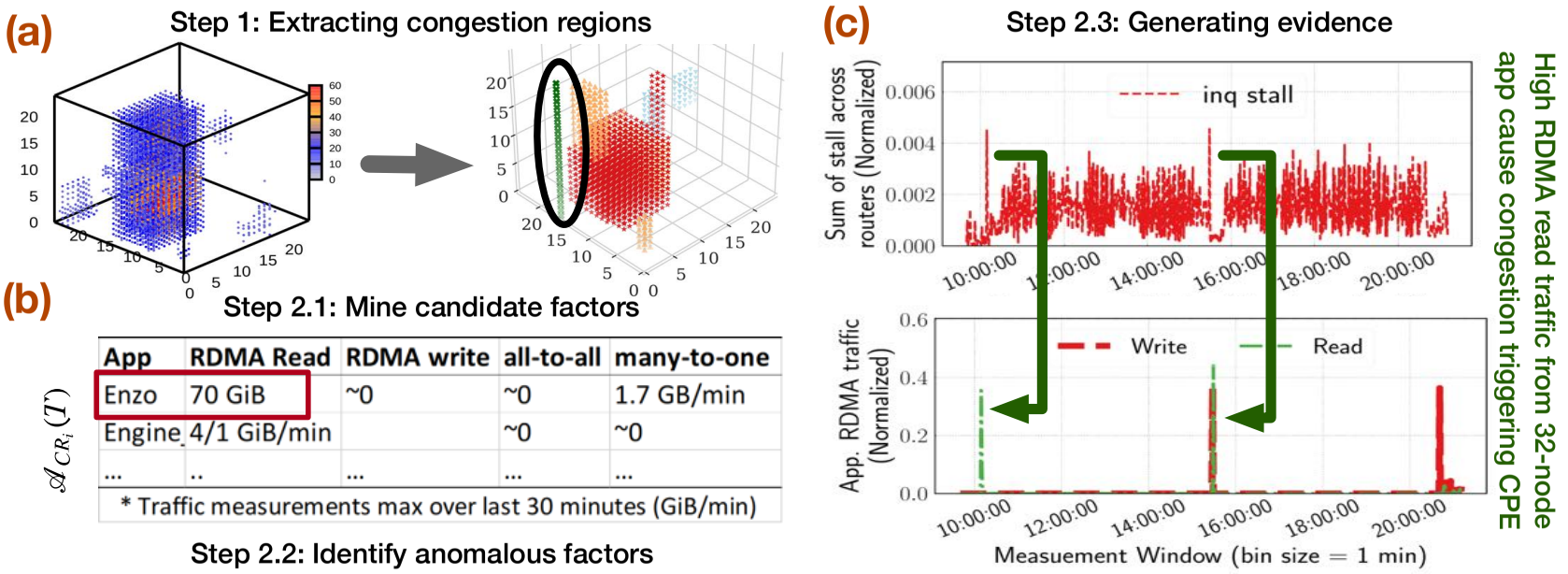


Figure 7.11: Detection and Diagnosis methodology applied to real-scenario

lapped with or were near the congestion regions) and network information to generate candidate factors that may have led to congestion. In this example, there were no failures; hence, this analysis generated only application-related candidate factors $\mathscr{A}_{CR_i}$, as shown in Fig. 7.11.

Step 2.2. Identifying anomalous factors. Next, we utilized the application traffic characteristics from candidate factors observed over the last 30 minutes (i.e., many-to-one or all-to-all traffic communication, and file system statistics such as read or write bytes) to identify anomalous factors by using a median-based outlier detection algorithm. In our example, as indicated in Fig. 7.11(b), the offending application was "Enzo" which was running on 32 nodes allocated along the "Z" direction at location (X,Y,Z) = (0,16,16) (indicated by a black circle in Fig. 7.11(a)). At the time of detection, "Enzo" was reading from the file system at an average rate of 4 GB/min (averaged over past 30 minutes and with a peak rate of 70 GB/min), which was 16x greater than the next-highest rate of read traffic by any other application in that time-window. The $\mathscr{A}_{CR_i}(T)$ for RDMA read bytes/min was 70 GB/min. The tool identified the RDMA read bytes/min of the "Enzo" application as the outlier feature. Hence, "Enzo" was marked as the anomalous factor that led to the congestion.

Step 2.3. Generating evidence. Once the potential cause had been established, further analysis produced additional evidence (e.g., distribution and correlation coefficient associated with link stalls in the congestion time window) to validate/verify the diagnosis results produced in Step 2.2. Fig. 7.11(c), in the top graph, shows a plot of the sum of stall rates on all links for all the Gemini routers local to the compute nodes used by the offending application, (i.e., Enzo) (normalized to the total stall rate throughout the duration of the application run). The two peaks (marked) in this top plot correspond to the increase in read bytes (normalized to total read bytes during the application run) shown in the bottom plot. Note that abnormal activity (an excessive amount of traffic to the file system) occurred around 10:10 AM (as shown Fig. 7.11(c)), which was about 20 minutes before the severe congestion developed in the system (seen in Fig. 7.11(a)). A "Medium" level of congestion was detected in the system spanning a few links (i.e., the congestion region size was small) at the time of the increased read traffic. Thus the cause was diagnosed to be "Enzo". Although, in this example scenario, the Cray congestion mitigation mechanism was triggered, it was not successful in alleviating the network congestion. Instead, the CR size grew over time, impacting several applications. "Enzo" was responsible for another triggering of the congestion mitigation mechanism at 3:20 PM (see the top graph in Fig. 7.11(c)). Monet detected and diagnosed it correctly.

7.8 RELATED WORK

There is great interest in assessing performance anomalies in HPC systems with the goal of understanding and minimizing application performance variation [293–295]. Monitoring frameworks such as Darshan [296], Beacon [297] and Kaleidoscope [48] focuses on I/O profiling and performance anomaly diagnosis. Whereas, our work focuses on assessing network congestion in credit-flow based interconnection networks. Typically congestion studies are based on measurements of performance variation of benchmark applications in production settings [293, 295] and/or modeling that assumes steady state utilization/congestion behavior [298–301], and thus do not address full production workloads.

There are research efforts on identifying hotspots and mitigating the effects of congestion at the application or system-layer (e.g., schedulers). These approaches include (a) use of application's own indirect measures, such as messaging rates [293], or network counters from switch that are accessible only from within an allocation [273, 302, 303], and therefore miss measurements of congestion along routes involving switches outside of the allocation; and (b) use of global network counter data [262, 304, 305, 249], however, these have presented only representative examples of congestion through time or executed a single application on the system [249].

In contrast, this work is the first long-term characterization of high-speed interconnect network congestion of a large-scale production system, where network resources are shared by nodes across disparate job allocations, using global network counters. The characterizations and diagnosis enabled by our work can be used to inform application-level [275] or system-level CEMRs (e.g., use of localized throttling instead of network-wide throttling). Perhaps, the closest work to ours is [306] which is an empirical study of cloud data center networks with a focus on network utilization and traffic patterns, and Beacon [297] which was used on TaihuLight [307] to monitor interconnection network inter-node traffic bandwidth. Like others, these works did not involve generation and characterization of congestion regions, diagnosis of congestion causes, nor a generalized implementation of a methodology for such, however, we did observe some complimentary results in our system (e.g., the existence of hot-spot links, the full bisection bandwidth was not always used, assessment of persistence of congestion in links).

Finally, for datacenter networks, efforts such as ExpressPass [243], DCQCN [308], TIMELY [309] focus on preventing and mitigating congestion at the network layer whereas efforts such as PathDump [310], SwitchPointer [311], PathQuery [312], EverFlow [313], NetSight [314], LDMS [262] and TPP [315] focus on network monitoring. These ap-

proaches are tuned for TCP/IP networks and are orthogonal to the work presented here. Our approach is complementary to these efforts as it enables characterization of congestion regions (hotspots) and identification of congestion causing events.

7.9 CONCLUSIONS AND FUTURE WORK

We present novel methodologies for detecting, characterizing, and diagnosing network congestion. We implemented these capabilities and demonstrated them using production data from NCSA's 27,648 node, Cray Gemini based, Blue Waters system. While we utilized the scale and data availability of the Blue Waters system to validate our approach, the methodologies presented are generally applicable to other credit-based k-dimensional meshes or toroidal networks. Our future work will involve extending the presented techniques to other network technologies and topologies.

CHAPTER 8: HPC: DOMAIN-GUIDED ML-DRIVEN FAILURE DETECTION & DIAGNOSIS FOR STORAGE SYSTEMS

Large-scale high-performance computing systems frequently experience a wide range of failure modes, such as reliability failures (e.g., hang or crash), and resource overload-related failures (e.g., congestion collapse), impacting systems and applications. Despite the adverse effects of these failures, current systems do not provide methodologies for proactively detecting, localizing, and diagnosing failures. We present Kaleidoscope, a near real-time failure detection and diagnosis framework, consisting of of hierarchical domain-guided machine learning models that identify the failing components, the corresponding failure mode, and point to the most likely cause indicative of the failure in near real-time (within one minute of failure occurrence). Kaleidoscope has been deployed on Blue Waters supercomputer and evaluated with more than two years of production telemetry data. Our evaluation shows that Kaleidoscope successfully localized 99.3% and pinpointed the root causes of 95.8% of 843 real-world production issues, with less than 0.01% runtime overhead.

8.1 INTRODUCTION

Large-scale high-performance storage systems frequently experience a wide range of *failure modes* [1, 36, 39, 316], including reliability failures (e.g., hang or crash) and resource overload-related failures (e.g., congestion collapse [317]). The net effects of these failures on systems and applications are often indistinguishable in terms of impact, and their mitigation strategies can vary significantly (e.g., throttling for congestion, or restart for a hung process). The inability to mitigate failures early enough can impact a single component (e.g., a data server), enable propagation of the failure across multiple interconnected components, or even cause a whole system outage, thereby adversely impacting application performance and resilience [44, 318, 251, 295, 37, 316, 319]. Thus, there is an need for not only detecting the failure, but also identification of the failure mode in real-time. As we show using a real-world failure scenario from the Blue Waters supercomputer's storage system (refer §8.2.1), a reliability failure can be construed as a performance problem and vice versa.

To address above problems, we propose Kaleidoscope, a system that uses machine learning (ML) to detect a failure, identify the failure mode, and diagnose the failure cause by using existing monitoring data in near real-time. Moreover, we have demonstrated Kaleidoscope and its scalability on Blue Waters, which is the largest university-based

high-performance computing (HPC) system in the world, in terms of both compute and storage nodes. We focus on high-performance storage systems because they have the most failures and lost compute hours[1]. For example, in 2018, NCSA reported that storage-related failures have accounted for 64.4% (i.e., >32 million core hours) of total lost core hours on a yearly basis. Further, the problem is expected to be worse in emerging and future exascale systems, with even lower mean time between failures and higher-impact service outages, because of increasing system scale, heterogeneity, and complexity [320, 321].

Why Machine Learning? Kaleidoscope uses multi-modal telemetry data from numerous monitors that provide system-wide temporal and spatial information on performance and reliability. The monitors either actively poll the system components [322, 323] (e.g., with pings/heartbeats), or passively aggregate performance and reliability measurements [297, 322, 262, 297, 324] (e.g., based on server load). The problem with telemetry data is that they are often noisy due to asynchronous collection [262, 324], failure propagation [37, 316, 325], and non-determinism in the system (e.g., in adaptive routing and load balancing) [326, 255, 327]. Therefore, when analyzed in isolation, telemetry data of a single modality may lead to misdiagnoses, i.e., false positives (e.g., in the case of failure propagation) and false negatives (e.g., in the case of partial failures). Moreover, the vast amounts of available telemetry data (on the order of TBs per day [328]) lead to cognitive overload of system managers [329]. They cannot keep up with the incoming data for identifying and debugging failure issues, significantly delaying the identification and mitigation of the failure.[2] To address those problems, Kaleidoscope uses ML methods that use domain-guided methods to accurately estimate the system state in the presence of noisy data, thereby detecting failures and identifying the failure mode and failure cause.

While existing approaches are useful [330–334, 297, 335–338, 40, 339, 294], they have significant drawbacks because they do not (i) jointly address reliability failures and resource-overload-related failures; (ii) focus on detecting and identifying failures and their failure mode in storage systems (except [334], which focuses on distinguishing network vs storage failures, and [322, 297], which mostly focuses on offline diagnosis); and (iii) deal with the difficulty of collecting/labeling training data, especially for rare failure scenarios in production settings [335, 336].

[1] In this chapter, we identify the failures at granularity of storage clients, network path to storage, storage servers, and RAID devices.

[2] For example, as we will show in §8.7.3, a partial failure of an I/O load balancer on Blue Waters, which was impacting application performance by as much as 25%, remained undetected for several weeks.

Our Approach. Kaleidoscope is a near real-time failure detection and diagnosis framework. It consists of hierarchical domain-guided interpretable ML models: (i) a *failure localization model* for identifying component failures (e.g., failures of compute nodes, load balancers, the network, storage servers, and RAID devices), and (ii) a *failure diagnosis model* for identifying the failure mode of a system component as either a resource-overload-related failure or a reliability failure.

The **failure localization model** uses ML and *I/O path-tracing data* to estimate the *failure state* of the storage components. I/O path-tracing data provide information on the route taken by the request (from the storage client on the compute node to the disk on the storage server) and the availability of the components on the route. The model incorporates the insight that the success of multiple I/O probes (e.g., a write I/O request) indicates that the components on the request path are healthy with a high probability. Each measurement in the I/O path-tracing data provides information on only a subset of storage components. Hence, the model jointly analyzes the I/O path-tracing data from multiple probes, and infers the probability of component failures.

To address the problem of noisy and multi-modal telemetry data and their joint analysis, our ML model uses the probabilistic graphical model (PGM) formalism to express the statistical dependence between the system components and the path-tracing data. Here, the failure state of each component is modeled as a *hidden* random variable; the path availability (i.e., the probability that an I/O request will complete successfully) is modeled as *observed* random variables; and the statistical dependence among random variables is derived using the design and implementation details of path-tracing monitors, the storage system, and the system topology. The proposed ML model is based on the insight that (i) even though individual path-tracing measurements might be noisy, (ii) groups of different measurements that are related to one another can be jointly considered to reduce the noise and estimate the failure state of the components, and (iii) the underlying statistical relationships between the storage components and the telemetry data can be used to correct for noise. We derive those statistical relationships by using the system topology and the paths taken by the I/O requests.

Although PGMs require less data for training and inference (compared to current approaches [336, 335]), dynamic collection of path-tracing data can be expensive due to intrusive instrumentation and data collection, which can interfere with application performance. To address that problem, Kaleidoscope uses *Store Pings*, a set of low-cost and low-latency probing monitors that not only probe a disk from a client by using an I/O request and record the response time (similar to `ioping` [340]), but also, unlike `ioping`, provide a mechanism for pinning (i.e., enforcing the use) of specific components on the

I/O request path (e.g., a disk, or data servers).

It is hard to distinguish between different failure modes because of limited observability, measurement noise, and failure propagation effects (described in §8.2.2). Notwithstanding, we have demonstrated that the proposed **failure diagnosis model**, which is a domain-informed statistical model, is able to accurately identify the failure mode and the likely causes (as discussed in §8.5.2) by using (i) components' telemetry data, which include performance metrics and RAS logs, and (ii) the failure state estimated using the failure localization model. The failure diagnosis model uses the Local Outlier Factor [52], an unsupervised anomaly detection method, which answers the question, "Which modality of the telemetry data (among RAS logs and performance metrics) best explains why one component is flagged as failed, while others are marked as healthy by the failure localization model?" The proposed model indicates the failure mode of the failed component as *either* a reliability failure (i.e., an error logs), or a resource-overload-related failure (i.e., a performance metric).

Results. We have implemented and deployed Kaleidoscope on the Cray Sonexion [341] high-performance distributed storage system of Blue Waters, a petascale supercomputer at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. Cray Sonexion uses the Lustre file system [342], which is used by more than 70 of the top 100 supercomputers [343] and is offered by cloud service vendors. The key results are as follows.

1. *High accuracy*: We used 843 production issues that were identified and resolved by the Blue Waters operators as the ground truth. Kaleidoscope correctly localized the component failures across all failure modes and resource overloads for 99.3% of the cases and accurately diagnosed the failure cause for 95.8% of the cases by pointing to the most likely failure cause and it distinguished between reliability failures and resource overloads/contention within 5–10 minutes of the failure incident. Moreover, Kaleidoscope found additional failures that were not present in the ground truth data, i.e., had not previously been identified.
2. *Low rate of false positives*: With respect to false positives, Kaleidoscope outperforms by $100\times$ the state of the art regression-based failure localization model, NetBouncer [330] customized for cloud networks, which focuses *only* on identifying partial and fail-stop failures and not on resource-overload-related failures and diagnosis.
3. *Low overhead*: The overhead introduced by Kaleidoscope is less than 0.01% of the system's peak I/O throughput.
4. *Long-term characterization*: Kaleidoscope was used to improve our understanding of storage-related failures by characterizing two years of production data.

8.2 BACKGROUND AND MOTIVATION

8.2.1 Blue Waters Storage Design

We describe the Cray Sonexion storage subsystem of Blue Waters and introduce our terminologies. Cray Sonexion is designed for large-scale HPC systems with I/O-intensive workloads, such as machine learning and large simulations. It's deployment on Blue Waters consists of 6 management servers, 6 metadata servers (MS), 420 data servers (DS), and 582 I/O load-balancers (LNET nodes). The storage servers in Cray Sonexion are connected via an internal Infiniband network (storage network). LNET nodes connect 28,000+ computing nodes (i.e., clients) on Cray Gemini interconnection network (compute network) to storage network. Cray Sonexion uses the Lustre parallel distributed file system to manage 36 PB of disk space across 17,280 HDD disk devices. The disks are arranged in a grid RAID [344] and are referred to as *object storage devices* (OSDs). Each storage server is attached to one or more OSDs. Lustre offers high-availability and failover features. In Lustre, data servers are arranged as active-active pair to achieve load balancing and high availability for connected OSDs, whereas metadata servers are arranged as active-passive pair for connected OSDs. The computing nodes are diskless: all I/O operations go by RPC to the LNET nodes, and the LNET nodes forward the request to the storage servers.

8.2.2 Motivating Failure Scenario

We describe a real-world failure scenario (see Fig. 8.1) which frequently occurs in the distributed storage system the of Blue Waters supercomputer to illustrate the difficulty of identifying the root cause of an application failure/slowdown using telemetry data. The telemetry data obtained during this failure scenario capture the following partial views:

1. *Storage view.* In this failure scenario, the telemetry data indicated high load and increasing service time on a pair of data servers. These data servers eventually hang and lead to unavailability of the files stored in these data servers. At the same time, other data servers (not shown in the figure) do not show symptoms of high load.
2. *Application view.* In this failure scenario, the NAMD [264] application issues *open* and *write* I/O requests. They are handled via FS clients (kernel modules) on each compute node. To write to the file, the FS client first *opens* the file and gets the file handler by accessing the *metadata server*, and then uses this file handler to directly *write* to the file on disk via the corresponding *data servers*. However, in this case, the *write* request fails

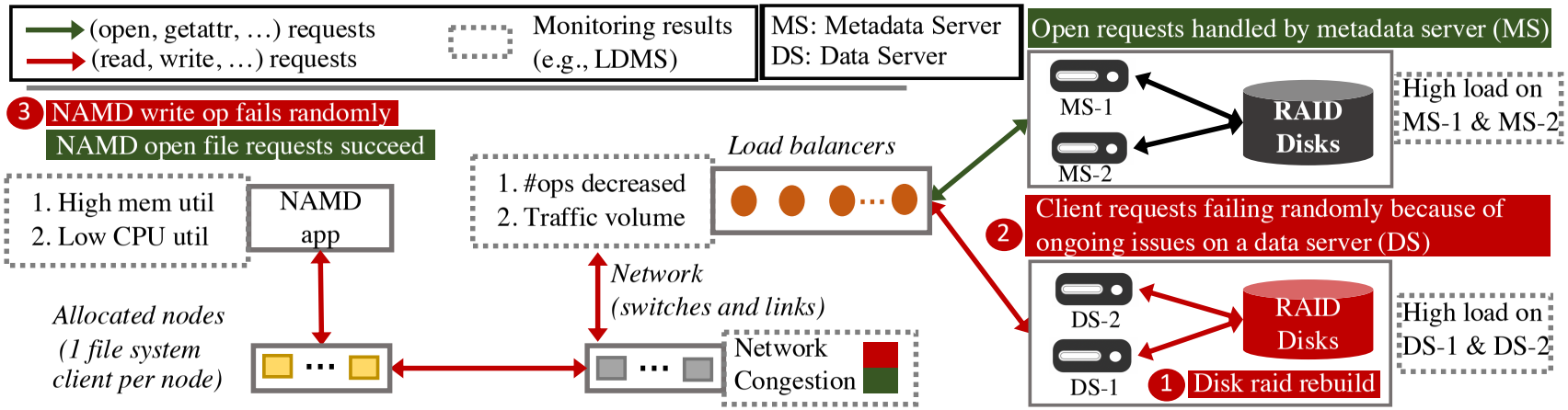


Figure 8.1: Propagation of I/O failure and challenges in identifying, localizing, and disambiguating the causes of I/O failures.

because of the FS client request timeout, despite the successful completion of the *open* request. The request failure causes the applications to fail. From the point of view of the application, the FS clients were partially failing.

Both views hint at a problem in the system, they are not sufficient for detecting and diagnosing the failure. The real cause of the failure was deeply hidden in the server logs. The analysis of the server logs revealed that a disk failure in the storage device (OSD in Lustre) was the real cause of the storage server and application failure. The failure of the disk triggers a RAID disk rebuild, which in turn decreases the effective I/O bandwidth available to two data servers (DS-1 and DS-2). The decrease in bandwidth causes an increase in the service time of I/O requests, which, in turn increases the load on data servers DS-1 and DS-2, which, in turn leads to server hang and unavailability of the files, ultimately causing application to fail. Intuitively, it can be seen from the failure scenario example that the failure mitigation will depend on both the failure location and mode. Overall, we find that the telemetry data, when analyzed in isolation and as illustrated in Fig. 8.1, provide outcomes and results that in general seem conflicting, even to experts. For example, the telemetry data on the application hint at high memory utilization, whereas telemetry data on the data server can hint at high load.

8.2.3 Challenges

The failure scenario above highlights multiple challenges: *Dataset heterogeneity & Fusion*. Large-scale HPC systems produce vast amounts of telemetry data (at application, network, and storage layers) by using multiple monitors across the system stack. These datasets are highly heterogeneous in nature (e.g., sampling frequency of monitors), and provides only partial observability into the system (i.e., storage and application levels). Thus, highlighting the need to jointly analyze datasets to avoid conflicting outcomes.

Data labelling and rare failures. There are challenges in both labeling the failure data, and acquiring them. This problem exacerbates due to a long tail of one-off, unique failures that are previously unknown and hard to anticipate based on historical data (discussed in §8.7.3).

Measurement uncertainty, noise, & propagation effects, emanated from (i) timing issues in asynchronous measurement and data collection intervals, (ii) non-determinism due to path redundancy and randomness in routing, and (iii) failure propagation leading to variability and noise in measurements.

Timeliness of analytics. Minimal number of monitors must be placed strategically across the system (i) to provide spatial and temporal observability, and (ii) to reduce data and

time required to perform analytics.

Those challenges make it difficult (i) to identify the failing component, and (ii) to discern the failure modes. That leaves system operators with no option but to comb through multiple monitoring dashboards to form their conclusions about failures based on their experience, and that significantly increases the response time for mitigating the impact of failure (upto 4–8 hours), leading to unexpected outages and impact. This is untenable for future exascale systems that would require realtime failure detection, diagnosis and mitigation.

8.3 KALEIDOSCOPE OVERVIEW

Fig. 8.2 shows the design of Kaleidoscope. The "Infrastructure" part (upper left) shows a simplified diagram of Blue Waters storage system (described in §8.7). The "Monitoring" part (lower left) shows the telemetry data collected from the system across the stack (described in §8.4). The "Hierarchical ML" part (upper right, described in §8.5) shows the interconnected ML models that provide failure localization (i.e., identifying the failed component), and diagnosis capabilities (i.e., identifying the failure mode and pointing to the anomalous telemetry data indicative of the failure). The "Outputs" part (lower right) provides an interpretable set of results and dashboards that can be used by the system managers (described in §8.6).

Kaleidoscope addresses the challenges of identifying failing components and discerning the failure modes described in §8.2.3 via the following approaches:

1. *Fusing heterogeneous telemetry data for increased observability.* Kaleidoscope uses telemetry data from across the system, capturing both the system and application views, to increase spatial and temporal observability. The fusion and comprehensive analysis of the data enable accurate detection of both resource overload and reliability failures.
2. *Hierarchical probabilistic ML models for dealing with data uncertainty and noises.* Kaleidoscope uses hierarchical probabilistic ML models that use domain knowledge to model measurement noises and failure propagation effects. The hierarchical ML models enable data analysis at different granularities and time scales.
3. *Unsupervised ML models for dealing with insufficient samples and rare failures.* Kaleidoscope uses unsupervised ML models and leverages domain knowledge on the system design and architecture to alleviate the challenges of (i) labeling the failures, and (ii) acquiring training data on rare failures, especially on rare one-off failures.
4. *Low-cost automation for timely analytics.* The use of unsupervised methods alleviates the need for costly training and re-training of models. Store Pings (refer to §8.4.1) are

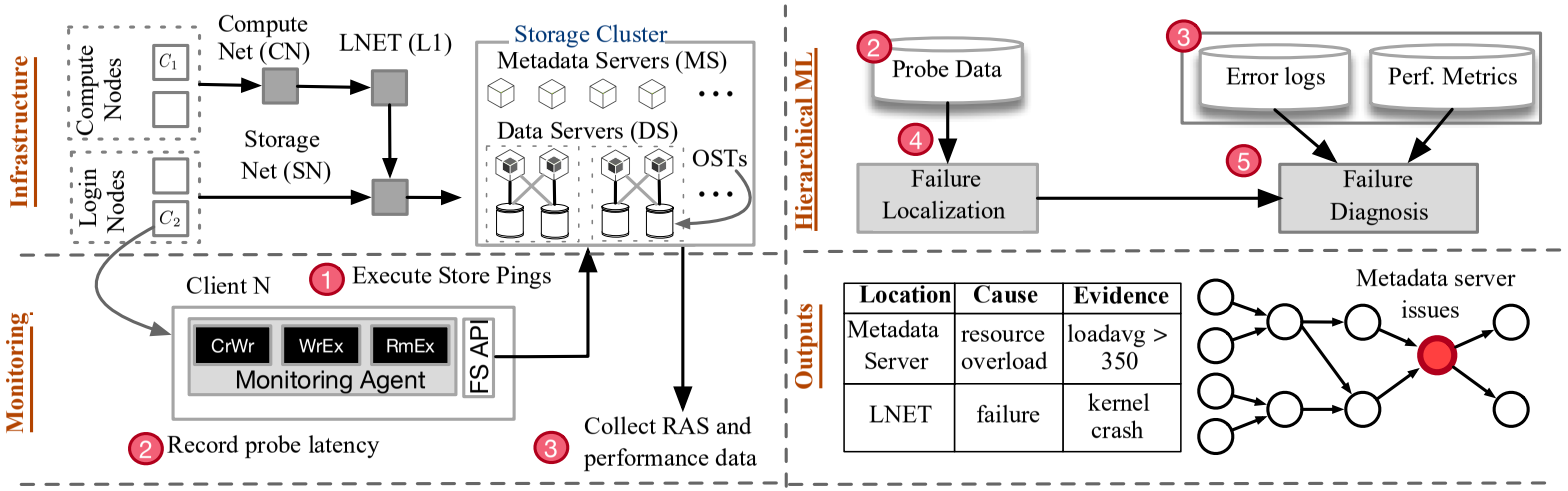


Figure 8.2: An overview of Kaleidoscope design and implementation.

low-cost monitor to provide observability into storage.

8.4 MONITORS & TELEMETRY DATA

8.4.1 End-to-end Probing Monitors

Kaleidoscope uses end-to-end I/O probing monitors (see ❶ & ❷ in Fig. 8.2), to collect path-tracing telemetry data. that provide observability into the health of each component on the path. For example, a successful probe from A to B through C and D reveals that all the components (A, B, C and D) are healthy; if the probe fails, it means that at least one component on the path is experiencing a failure. *A probe is marked as successful when it completes within a pre-specified time limit (i.e., meets its service-level objective); otherwise it is marked as failed*. Although distributed path-tracing tools exist (e.g., Zipkin [345] and Uber Jaeger [346] for microservices, and Darshan [347] for HPC I/O, dynamic collection of tracing data can be hugely costly. Moreover, the available tools only provide application views and fail to provide observability into the storage infrastructure view, which is critical, as we show in §8.2.2. Hence, we created Store Pings, which are low-cost probing monitors that not only probe a disk from a client by means of I/O requests (similar to `ioping` [340]) and record the response time, but also provide a mechanism for pinning the path of the I/O requests to a disk through specific load balancers and servers. The pinning of the path eliminates the need for tracing of the request, and thereby reduces the overhead of data collection on path availability. While Store Pings are analogous to the ICMP-based network `ping` (which provides visibility only into the network), the two are significantly different. Specifically, Store Pings are designed for storage systems and provide visibility across the entire system stack, which includes compute, network/interconnect, and storage subsystems. Since, Store Pings generate an I/O probing request of fixed size, an **I/O failure** occurs when the I/O completion time is higher than or equal to one second. We use one second as SLO because 99% of the Store Ping probes on Blue Waters completes within one second.

Path Pinning. Store Pings provide path-pinning capabilities by leveraging Lustre's file system support for pinning of a file on a specific object storage device (and hence the data server),[3] thereby eliminating the need to modify Lustre to support path pinning. Since the metadata server has all the data chunk information, an I/O request to the file uniquely identifies both the OSD and the data server. It also prunes the number of possible paths

[3]Store Pings executes independently of other applications. It creates and operates on its own set of files to achieve the monitoring goals.

that can be taken by the I/O request (from the client to the OSD). For example, a Store Ping executing on a compute node (which is a storage client) and accessing data on an OSD can use only 4 load balancers (LNETs) instead of all the LNETs (of which there are more than 500) in the system. Although pinning of all the components (e.g., pinning of I/O requests to a particular LNET) on the path is desirable, it is unnecessary and would require changes in the proprietary software and hardware in the compute and storage system to support deterministic routing. We leverage probabilistic models to handle non deterministic paths (§8.5).

Increased Observability. The API of a Store Ping is: `store_ping(ost, *io_op, kwargs)`, where `*io_op` is a function pointer to an I/O operation, and `kwargs` is the argument of `*io_op`. Store Pings use direct I/O requests to avoid any caching effect, which ensures that each I/O request traverses all the way from the clients to the disks on the data servers. We designed three types of Store Pings, `CrWr`, `WrEx`, and `RmEx`, which correspond to three different I/O requests: (i) `CrWr`, which creates and writes a new file; (ii) `WrEx`, which writes to an existing file; and (iii) `RmEx`, which removes an existing file. `CrWr` and `RmEx` test the functionality of the metadata servers, whereas `WrEx` tests the functionality of the data servers (and, correspondingly, RAID disks). For example, a `CrWr` requires two different back-end operations to complete: (i) creation of a file by a metadata server on a random data server (and the corresponding RAID disks) and addition of the file entry to the metadata index, and (ii) opening and writing of a file on the data server (and the corresponding RAID disks). The payload of a write request is only 64 bytes. Together, the three types of Store Pings test all the storage subsystems (which include storage clients located on compute nodes, network interconnections, storage servers, and RAID devices) that are involved in ensuring successful I/O operations.

Placement. Store Pings are strategically placed in the system to provide both spatial and temporal differential observability in near real-time. Store Pings generate probing requests continuously at regular intervals to measure the availability and performance of storage components. Note that Store Pings should be enabled only on a subset of clients to reduce the overhead of the Store Pings and their impact on existing I/O requests, while providing complete spatial observability.

Selecting the number of Store Pings and their placement can be formulated as a constraint optimization problem. The subsets of components that can be tested together are limited by the set of I/O paths, which are in turn limited by the topology, probing mechanism, and I/O request routing protocols. We use *Boolean network tomography principle* to solve the constraint optimization problem of selecting the number of Store Ping monitors

and their placement [348]. Specifically, the placement of monitors[4] in Kaleidoscope is guided by the *sufficient identifiability condition* (discussed in [348, 349]), which states that in a topology graph G of a system (in this case the Lustre storage system) consisting of both monitor and non-monitor nodes, any set of up to k failed components is identifiable if for any non-monitor $v \in G$ and failure set F with $|F| \leq k$ such that $v \notin F$, there is a measurement path going through v but no node in F. In other words, there must exist a set of I/O paths that can be used by Store Pings to uniquely identify the failure-state of each component and detect up to k concurrent failures. This is also referred as *spatial differential observability* and allows us to handle redundancies as long as the condition is met. [348] provides set of rules and algorithms to meet sufficient identifiability condition to identify number of monitoring nodes and their placement for any arbitrary storage system. We omit the detailed discussion because of lack of space. Blue Waters's system managers not only want to identify failures of storage components but also failures of service nodes and login nodes. We place Store Ping monitors on storage clients that (i) have different system stacks (e.g., kernel versions), (ii) are physically located on different networks, and (iii) execute different services (e.g., scheduling, user login, and data moving). Specifically, we place monitors on all the service nodes that provide scheduling and other capabilities (64 nodes); import/export (I/E) nodes (25 nodes) that move bulk data into and out of the storage system; and login nodes (4 nodes), which launch applications. The I/E nodes and login nodes are on the storage network, whereas the service nodes are on the proprietary compute network fabric. This placement scheme meets both the production requirements (given by system managers) and theoretical requirements (from network tomography principle).

Probing Plan. At any given time, Store Pings are executed from (i) all login nodes, (ii) 1 out of 64 service nodes chosen randomly, and (iii) 1 out of 25 I/E nodes chosen randomly. That probing plan satisfies our minimal probing plan for inferring storage system health, while providing reliability for the monitoring infrastructure; if a client failure occurs, another client can be chosen as a monitor. Store Pings are executed every minute for each OSD, data server, and metadata server. That results in 72 `CrWr` and 72 `RmEx` (from 6 clients to 6 metadata servers and 6 OSDs) and 5,184 `WrEx` (from 6 clients to 432 data servers and 432 OSDs) requests per minute.

8.4.2 Component Logs

Kaleidoscope uses a comprehensive monitoring system (similar to the monitoring

[4]In the network tomography formalism, both the ends of the probing path is referred as monitors. A Store Ping path starts at a storage client and ends at an object storage device (OSD).

system described in [350, 297]) to collect performance measurements and RAS (reliability, availability, and serviceability) logs for each system component (including compute nodes, load balancers, network switches, and storage servers) in real-time (see ❸ in Fig. 8.2). We use the Light-weight Distributed Metric Service (LDMS) [262], a data-aggregation tool, to collect performance measurements (e.g., `loadavg`, memory utilization, disk latency) for compute nodes, load-balancers (LNETs) and switches. We use ISC (the Integrated System Console) [351] to collect performance measurements on storage components (e.g., disks, and servers), LDMS data, and RAS logs on a centralized server.

8.5 HIERARCHICAL MACHINE LEARNING MODELS

Kaleidoscope uses hierarchical domain-guided unsupervised ML models to provide live forensics capabilities. These hierarchical ML-models include: (i) failure localization model (for identifying the failed nodes), and (ii) failure diagnosis model (for identifying the failure mode of the failed node).

8.5.1 Failure Localization Model

Kaleidoscope uses a *failure localization* model (see ❹ in Fig. 8.2) for identifying component(s) that are failed or overloaded, and thus are leading to I/O failures. Kaleidoscope uses telemetry data obtained from Store Ping monitors for that purpose. However, Store Ping measurements are noisy (due to asynchronous data collection, adaptive routing/load-balancing, and failure propagation among others) and provide partial view (i.e., the measurements only provide information on a subset of the system components). These challenges are hard to deal with traditional threshold or voting-based methods which often lead to over-counting and misdiagnosis [330]. Therefore, we model these noise/uncertainties in the telemetry data as well as provide a formalism to fuse these partial views.

We use probabilistic graphical model (PGM) formalism, in particular the factor graph (FG) model [352], to jointly analyze and fuse the telemetry dataset from all the Store Pings monitors placed on the system, while accounting for the noise and related uncertainties. PGMs specify the relationships between the random variables using a graphical structure, where a node represents a random variable, and an edge represents the statistical relationship between random variables. This graphical structure allows PGMs to capture complex conditional independence between the random variables (i.e., domain

knowledge), specified in a human interpretable manner. Using such domain knowledge in turn reduces both the data requirements (compared to supervised machine learning methods [334, 330, 336]) as well as inference time. The proposed PGM model is based on the insight that even though individual Store Ping measurements might be noisy, groups of different Store Ping measurements that are related to one another can be jointly considered to reduce the measurement errors, all while estimating the failure state of the components. Kaleidoscope uses the most general form of PGMs called factor graphs (FGs), which is a generalized formalism for specifying and computing inference on PGMs. In our FG model, the *failure state* of each component (which is *hidden*) on a path and its corresponding path availability (which is *observed* using Store Ping telemetry data) are specified as random variables, and the functional as well as statistical relationship between hidden and observed variables as (in terms of path) factor functions. An inference on the FG model allows Kaleidoscope to estimate failure state of each component, and explain the observed telemetry data. This determination of the failure state localizes failed components in the system.

Formalism. We define the *health*, and hence the failure state, of a component as as random variable, $X_i^{(t)}$, whose value captures the probability of a component i successfully serving an I/O request at time t. We use the shorthand X_i for $X_i^{(t)}$, as the variable changes at every time step.[5] In the absence of measurements, X_i is derived from a prior beta distribution[6], i.e., $X_i \sim Beta(\alpha, \beta)$, where α and β determine the shape of the distribution. At any time step, α and β are updated based on the inference at the previous time step (described later in the 'inference' paragraph).

Store Ping-based monitoring provides *reachability* measurements between a client C_i and an OSD OSD_j. We use a random variable $Y_{\langle C_i, OSD_j \rangle}$ to denote the number of successful Store Pings between C_i and OSD_j in the interval $(t-1, t]$. We model $Y_{\langle C_i, OSD_j \rangle}$'s prior using a binomial distribution, $Y_{\langle C_i, OSD_j \rangle} \sim Binomial(A_{\langle C_i, OSD_j \rangle}, N)$, where $A_{\langle C_i, OSD_j \rangle}$ denotes the reachability from the C_i to OSD_j, and N denotes the total number of Store Pings issued from the C_i to OSD_j. We use binomial distribution because it allows us to compute the probability of observing a specified number of "successes" (in this case, number of successful Store pings between C_i and OSD_j), which we observe through our telemetry data.

[5] All the variables defined below are time variant, however we use the same shorthand to simplify the description.

[6] Beta distributions are: (1) continuous distribution which models the success of an event (here an I/O request) and (2) commonly used as a conjugate prior for Bernoulli and Binomial random variables (which we use in our model). Moreover, use of conjugate priors drastically reduces the computation time for inference [353].

We use the domain knowledge of underlying statistical relationships between the telemetry data and the components' health to calculate $A_{\langle C_i, OSD_j \rangle}$. These statistical relationships are based on the understanding of system topology and I/O request path. For example, lets consider the case when the exact routing information of an I/O request is available using a path tracing tool. Using the route information, we could have determined $A_{\langle C_i, OSD_j \rangle}$ solely by the product of individual component's health (i.e., all components on the path must work for the request to be successful): $A_{\langle C_i, OSD_j \rangle} = \prod_{i \in \mathbb{P}(\langle C_i, OSD_j \rangle)} X_i$. Where $\mathbb{P}(\langle C_i, OSD_j \rangle)$ denotes the path between C_i and OSD_j.

Recall from §8.4.1, that collecting path-tracing data is expensive. Hence, we must model the redundancies (e.g., high availability pairs and failover) and non-determinism to calculate $A_{\langle C_i, OSD_j \rangle}$. In our system, a Store Ping destined for an OSD may take a different path among several possible paths depending on the load and routing policies. For the sake of clarity, we illustrate the procedure to model redundancies by modeling the path of I/O request through a high-availability data server. We use the same methodology to model other redundancies (e.g., load-balancers and network paths). An I/O request to an OSD can be routed through one of the two data servers connected to it. Hence, a destination OSD is not reachable if both data servers (DS-1 and DS-2) connected to it are unavailable or the OSD itself is not available. R_{OSD_i}, the probability of an I/O request completing successfully from a load balancer to an OSD_i, is given by eq. (8.1).

$$R_{OSD_i} = (1 - (1 - X_{DS_1}) \cdot (1 - X_{DS_2})) \cdot X_{OSD_i} \tag{8.1}$$

In eq. (8.1), X_{DS_1} and X_{DS_2} denote the health of data servers in the HA pair associated with the OSD (denoted by OSD_i). In the equation, $1 - X_{DS_1}$ and $1 - X_{DS_2}$ determine the probability distributions of the DS_1 and DS_2 to be *failed* respectively, and their product determines the probability distribution that both will be in a failed state. That probability distribution, when multiplied by the probability distribution of the OSD's health, gives the reachability of the OSD from one of the data servers. As shown in Fig. 8.3 (bottom half), the $A_{\langle C_i, OSD_j \rangle}$ between client C_i and OSD_j is given in eq. (8.2).

$$A_{\langle C_i, OSD_j \rangle} = X_{C_i} \cdot X_{L_a} \cdot X_{CN_b} \cdot X_{SN_c} \cdot X_{MS_d} \cdot R_{OSD_j} \tag{8.2}$$

In eq. (8.2), C_i, L_*, CN_*, SN_*, MS_*, and OSD_j stand for *client*, *LNET*, *compute network*, *storage network*, *metadata Server*, and *object storage device* respectively, as shown in Fig. 8.3. Here, the path availability $A_{\langle C_i, OSD_j \rangle}$ only models the non-determinism associated with load balancing on the data server. We follow a similar approach to derive $A_{\langle C_i, OSD_j \rangle}$ for

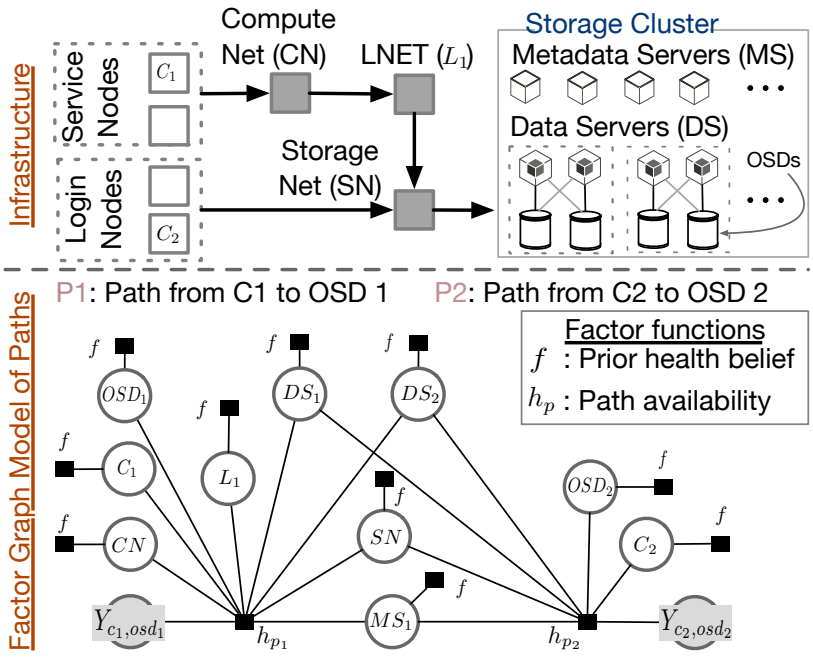


Figure 8.3: The FG model for failure localization. Non-shaded circles represent hidden random variables, and shaded circles represent observed random variables (measurements).[7]

our system. Moreover, Kaleidoscope models the temporal evolution by using estimated component health parameters from previous inferences and uses the uncertainty to quantify the confidence in the inference results.

The model described above can be represented using a factor graph (FG) that models the relationship between different random variables (shown as circles in Fig. 8.3) and *functional relationships* known as factor functions (shown as dark boxes). The relationships between random variables are extracted from the system topology diagram, which can be derived from the reference manuals or mined using tracing tools.

Fig. 8.3 shows a part of the FG that models (i) the health of components that lie on the path of $\langle C_1, OSD_1 \rangle$ and $\langle C_2, OSD_2 \rangle$, and the path availability for these components. The components OSD_1, OSD_2, DS_1, and DS_2 form a high-availability (HA) pair (i.e., a I/O request to a particular OSD in the pair can be served by either of the data servers). The circles in the FG represent random variables (e.g., a component's health). The factor functions, represented by squares, encapsulate the relationships among the random variables. The singleton factor functions f_i encapsulate the prior belief on the health of the component (which is known from a previous time step or from training time), which is given by

[7]Only paths from $\langle C_1, OSD_1 \rangle$ and $\langle C_2, OSD_2 \rangle$ are shown, for clarity. Redundancies and network components have also been removed for clarity.

the beta distribution (described above). The multivariate factor function $h_{\langle C_i, OSD_j \rangle}$ models the number of successful Store Pings on a path, given by the binomial distribution (described above).

Inference. With the factor graph model, we can calculate the health of each component X_i in the system. The expected health of a component i can be estimated as $E[X_1, X_2, X_3, ...|Y_{p_1}, Y_{p_2}, Y_{p_3}, ...]$. Observations ($Y_{p_1}, Y_{p_2}, Y_{p_3}, ...$) and the prior belief on the health of components (α and β for each X_i) are needed at time step T_j. Y_{p_i} is measured as the number of observed successful Store Pings during a specified interval, and α and β are obtained from the inference result at the previous time step, and at time zero initialized to 0.5 (i.e., there is no prior information of the components being either healthy or failed.). Intuitively, the inference procedure biases the prior belief of the model on the failure states of the components using the telemetry data (i.e., Store Ping probing data) obtained in the current time step. Kaleidoscope solves the inference task by using the Monte Carlo Markov Chain algorithm [354], a technique for estimating the expectation of a statistic from a complex distribution (in this case, $E[X_1, X_2, X_3, ...|Y_{P_1}, Y_{P_2}, Y_{P_3}, ...]$) by generating a large number of samples from the model and directly estimating the statistic. We also quantify the confidence in the inference results and use it to reduce the false positives. Our model declares a component to be failed *only* when the confidence in the inference is more than 75%. Failure localization model is implemented using PyMC3, a Python-based probabilistic programming language [355]. It uses samples collected over five minutes, i.e., the results of 26,640 I/O requests, for inference.

Training. Note that training is not explicitly required for the proposed model. However, it can help bootstrap the model before deployment. One key advantage of using probabilistic models like FGs is that training of such models can be reduced to inference on the model parameters (i.e., estimating the parameters of the used probabilistic distributions). In the case of a parametric FG that parameterizes certain statistical relationships (as in our model), we set up the training problem just like the inference problem to pick the set of parameters that can explain a data trace generated by the system.

8.5.2 Failure Diagnosis Model

The *failure diagnosis model* (see ❺ in Fig. 8.2) leverages (i) components' telemetry data, which include performance metrics and RAS logs, and (ii) the failure state estimated with the failure localization model, to understand the likely cause of the failure. It uses the insight that a failed component behaves significantly differently from its healthy counterparts. For example, telemetry data obtained from a failed data server may reveal high

load (e.g., high memory utilization) or an error (e.g., process crash), whereas the telemetry data of the healthy data servers will not reveal any such failures.

We use that insight to formulate the failure-diagnosing problem as an explainability problem that can be phrased as a conditional question: "Which modality of the telemetry data (amongst RAS logs and performance metrics) best explain the reason why one component is flagged as failed while others to be marked as healthy by the failure localization model?"

The failure diagnosis model answers that conditional question by statistically comparing the measurements of the failed component and the healthy components by using an unsupervised ML-based anomaly detection method that selects a measurement that best distinguishes the failed components from the healthy ones. If there has been a reliability failure (e.g., `kernel crash`), it will point to error logs, and if there has been a resource-overload-related failure, it will point to a performance metric, such as `high server load`. Note that the conditional question is fundamentally different from the non-conditional question "Which modality of the telemetry data are anomalous across all components?" The non-conditional question usually suffers from noises (e.g., each component produces hundreds if not thousands of error logs that may not be relevant to diagnosing the failure [339]), making it challenging to precisely distinguish anomaly from normal behavior. In other words, the conditional question eliminates the noise in the first place. For example, we should not flag a data server as failed just because its utilization is higher than the other servers. However, if the *failure localization model* identifies the server as failed and high load is the only factor that differs the failed data server from the other healthy data servers, then the failure of the failed data server is most likely due to high load.

Diagnosing Reliability Failures. Kaleidoscope attributes and diagnoses reliability failures based on log analysis. Working with the vendor and national labs, we have curated a library of regular-expression patterns to filter error logs that are indicative of reliability failures (e.g., `kernel dump`). Currently, our library consists of 184 regular expression patterns. In the absence of such a library, we could use existing log pattern mining tools (e.g., Baler [356]) to automatically create a library of regular-expression patterns from existing logs, and then filter the patterns based on their severity level, i.e., by using patterns of a severity level of 4 (`warning`) and above.

Kaleidoscope filters RAS logs of storage components by using the library of aforementioned regular-expression patterns (§8.4.2). The error logs generated by the failed/failing components are compared to the error logs of healthy components, $\delta = L_{\text{UO}} - \bigcup_{i \in HO} L_i$,

where L represents the log set, and UO and HO represent failed/failing and healthy components, respectively. If $\delta \neq \varnothing$, then δ is provided as evidence, and the failed status is attributed to component failures.

Diagnosing Resource Overload and Contention. Kaleidoscope attributes and diagnoses resource overload/contention based on the following telemetry data: (i) the server performance metrics (e.g., `loadavg`), which captures the load of a server at 5-minute intervals; (ii) the RAID device performance metrics (e.g., `await` time, which captures the average disk service time (in milliseconds)), and taken by a disk device to serve an I/O request; and (iii) the network performance counters (e.g., stall).

Kaleidoscope compares the performances of storage components of similar types (e.g., data servers) by using the local outlier factor (LOF) algorithm [52]. The LOF is based on the concept of *local density*, where locality is given by k-nearest neighbors and the density is estimated by the distance to the neighbors. By comparing the local density of a target with the local densities of its neighbors, Kaleidoscope identifies regions with similar densities, and pinpoints outliers that have a substantially lower density than their neighbors in terms of performance metric values. Using the LOF algorithm, we calculate LOF score using the aforementioned telemetry data for each component indicating the similarity/dissimilarity of the component to other components in terms of its performance. Using that score, we can ask the aforementioned conditional question. If we find that the failed component has a score of 1.0 (i.e., the performance is similar to that of other components), then there is no reason to believe that the component failure was caused by a resource overload/contention problem.

We chose LOF because storage components within a homogeneous group could have different modes of operations that are not indicative of anomalies. For example, we found normal states in which k data servers had a low `loadavg` (less than 10) and $N - k$ data servers had a high `loadavg` (larger than 64). However, if there is one data server with a `loadavg` significantly higher than that of the rest, it indicates an anomaly, and such behavior is effectively captured by LOF. In Kaleidoscope, we use a configuration named LOF_r and declare a component to have "resource overload/contention" if the LOF value of the failed component is LOF_r times larger than the max LOF value of a healthy component. (The default value of LOF_r is 1.5.)

We use the outlier-based method to ask the conditional question for their simplicity and effectiveness. Our approach is very similar to that of, and inspired by, Distalyzer [339]. However, Distalyzer is only suited to offline diagnostics as it does not provide a methodology for identifying/labeling failed components because it assumes that such a label is already available. Thanks to Kaleidoscope's hierarchical approach, it is

Table 8.1: Effectiveness (measured by true positives) of Kaleidoscope's triage and root-cause analysis.

Localization	**True Positive**	**False Negative**	**Total**
	837 (99.3%)	6 (0.7%)	843
Diagnosis	**Correct Diagnosis**	**Misdiagnosis**	**Total**
Reliability Failure	340 (98.3%)	6 (1.7%)	346
Overload/Contention	468 (94.2%)	29 (5.8%)	497

possible to integrate more sophisticated statistical methods and log analysis methodologies [339, 297, 357, 40].

Training and Inference. Failure diagnosis is completely unsupervised, and therefore does not require any training. However, the method requires a library of regular-expression patterns that is created in the offline mode through manual methods (using vendor support) or automatic methods (using statistical learning techniques such as clustering [356, 358]). Failure diagnosis is implemented in Python.

8.6 EVALUATION

We have deployed Store Ping monitors on Cray Sonexion for two years and Kaleidoscope's live forensics on Cray Sonexion for more than three months. However, to comprehensively evaluate the effectiveness of Kaleidoscope's live forensics, we fed the two years of monitoring data collected by Store Ping monitors *retrospectively*. The evaluation is based on 843 production issues resolved by the Cray Sonexion operators over the two-year span. Each of the 843 issues has a corresponding report after manual investigation. We use the dataset as the *ground truth* to measure the true positives and false negatives. We also quantify the false positives by inspecting 100 randomly selected issues from the issues reported by Kaleidoscope.

8.6.1 Effectiveness

Kaleidoscope observed 26,596 I/O failure events in total (25,427 resource overloads and 1,169 reliability failures). The number is significantly higher than the 843 production issues. This is because many of the I/O failure events are transient and short-cycled and thus does not lead to production issues. In Cray Sonexion, operators use the following two policies to identify important I/O failure events for manual investigation:

1. certain class of failures are auto-fixed by the system within one minute of occurrence (e.g., network recovery to route out bad links). Kaleidoscope finds out these cases and stops alarms by monitoring recovery events.
2. resource overload/contention events are often transient in nature and a mitigation action is triggered only when the condition continues for more than 30 minutes. Fig. 8.4 shows the histogram of the duration of these I/O failures.

Applying the above two policies on the results generated by Kaleidoscope reduces I/O failure events from 26,596 to 1,525. We evaluated the effectiveness of Kaleidoscope regarding its accuracy of both localizing the failed components and diagnosing their root causes. Table 8.1 summarizes the results.

Localization accuracy. Kaleidoscope was able to localize the failed components (caused by either reliability failures or resource overload/contention) for 99.3% of the production issues (837 out of 843). Only six out of 843 production issues were not detected by Kaleidoscope. We read the report and found that none of the six issues had any impact on the I/O completion time. All six issues belonged to disk drive failures. Those failures were recorded and flagged for repairs to avoid RAID failures. Kaleidoscope additionally detected 688 events. We refrained from labeling these additional events as false positives because there was no evidence supporting that these were not actual issues. On the contrary, we found that many of the performance issues either went unnoticed because the system was not monitored adequately (such as no dedicated monitoring for disk load), or were ignored because there was no automatic alerting mechanism to take remediation action on the events in time.

Diagnosis accuracy. Among the 843 production issues, 346 were caused by reliability failures and 497 were caused by resource overload. Applying the same heuristic on Kaleidoscope output as used by the operators (described above), we found that Kaleidoscope reported 340 reliability failures and 468 overloads, which accounts for 98.3% of reliability failures and 94.2% of the resource overload/contention issues from the list of production issues (see Table Table 8.1). Kaleidoscope additionally detected 22 reliability failures and 558 resource overload issues. We had managed to manually validate 100 of those resource overload issues detected by Kaleidoscope and they were indeed true. Kaleidoscope presented error logs or performance metric to the operator for further investigation. Kaleidoscope diagnosis module missed 35 production issues: (i) 6 issues were missed by localization module, and (ii) 29 resource overload issues coincidentally had random noise in the logs, which confused Kaleidoscope.

False Alarms & Misdiagnosis. It was challenging to measure false positives (FP) due to the lack of ground truth dataset—an I/O failure detected by Kaleidoscope but not being

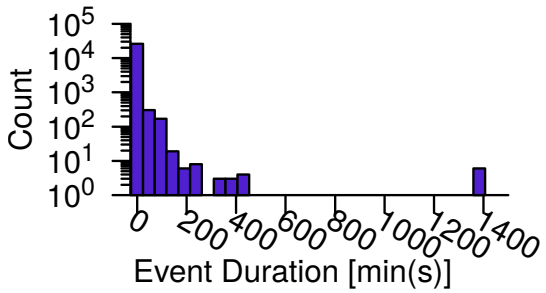


Figure 8.4: Histogram of failure duration.

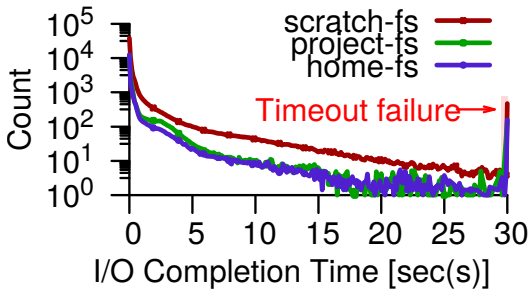


Figure 8.5: `WrEx` measured latency on three file partitions.

resolved could come from non-technical reasons (e.g., low priority jobs).

To statistically estimate the FP rate, we randomly selected 100 failures identified by Kaleidoscope (referred as Kaleidoscope events): 50 tagged with "reliability failures" and 50 tagged with "resource overload/contention." Kaleidoscope's failure localization model was able to localize *all* true cases of failures correctly. However, Kaleidoscope's failure diagnosis model misdiagnosed the root cause of four (out of 100) cases.

8.6.2 Baseline Comparison

Kaleidoscope is the first (to our knowledge) system that supports real-time forensics for peta-scale storage systems. In our work, we compare Kaleidoscope with NetBouncer [330]. We choose NetBouncer because it significantly outperformed existing failure localization methods designed for large-scale networks [359–361] and was tested on a real deployment.

Table 8.2 shows the localization accuracy of Kaleidoscope and NetBouncer [330], the state-of-the-art failure localization method. Our implementation was reviewed by the author(s) of NetBouncer. NetBouncer has 110 true positives (out of 186 true positive cases found in 6 months of our retrospective data), i.e., it misses 76 true cases that were captured by Kaleidoscope. NetBouncer's missing those issues because it is incapable of modeling 1) non-determinism due to path redundancy and 2) temporal evolution of the component state, which is modeled by Kaleidoscope as discussed in §§8.5.1. Furthermore, Kaleidoscope reports a total number of 4,892 events, far less than the number reported by NetBouncer. Given that self-recovered failures and overload condition less than 30 minutes can be filtered out, we can reduce the alarms to 412 (instead of 4,892) and 92,000 (instead of 116,072) respectively. The significant difference in the results of NetBouncer and Kaleidoscope is due to NetBouncer's inability of distinguishing I/O failure events as reliability failures or overload/contention.

Table 8.2: Comparing failure localization in Kaleidoscope and NetBouncer using 6 months of production data consisting of 186 issues.

	True Positive	**False Negative**	**Alarms**
Kaleidoscope	184	2	4892
NetBouncer	110	76	116,072

Table 8.3: Impact of 100 Store Ping monitors running at 30 second interval on IOR benchmark [362]. The mean value of I/O throughput without Kaleidoscope is normalized to 100. The off configuration is shared across both 100 and 6 montiors.

Kaleidoscope	**100 monitors**		**6 monitors**	
	Mean	*Std*	*Mean*	*Std*
Off	100	0.15	100	0.15
On	97.58	0.32	99.99	0.12

8.6.3 Monitoring Overhead

We used the IOR benchmark [362] to measure the monitoring overhead in a worst-case scenario. The measurement used stress testing to max out the throughput offered by Cray Sonexion. IOR was running on 4,320 compute nodes during this measurement. Table 8.3 shows the monitoring overhead introduced by Store Pings when (i) 100 monitors were running at 30 second interval and (ii) 6 monitors were running at one-minute interval. Recall from §8.4.1, we need 6 monitors for our probing plan to provide sufficient measurements, and we show result for 100 monitors to show the scalability of our solution. Store Pings decreased mean throughput *only* by less than 0.01% in Cray Sonexion. However, scaling to 100 monitors and increasing the frequency by $2\times$ would decreases the throughput by less than 2.42%. Note that the average throughput in production is significantly below the peak throughput under the stress test. We also measured the time difference between the launch of Store Pings for a given interval and found that all Store Pings were launched within 10 seconds and 98.4% were launched within 3 seconds.

8.7 OPERATIONAL EXPERIENCE

Our interaction with Cray Sonexion's operators shows that Kaleidoscope help them understand the tail latency and performance variation in near real time. Operators can detect performance regression by comparing the measurements from different points of time. Fig. 8.5 shows the latency measurement histogram for the `WrEx` Store Pings (`RmEx` and `CrWr` are omitted for clarity). We can see that 99% of `WrEx` completed within one

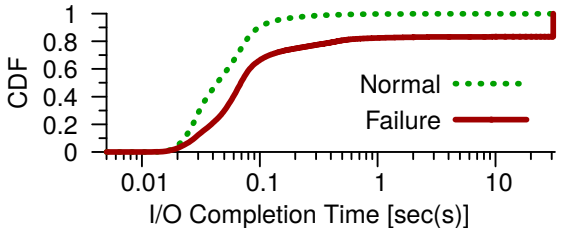


Figure 8.6: CDF of I/O request completion time under reliability failures ("Faliure") and no failures ("Normal").

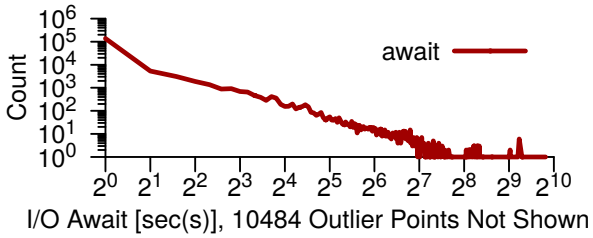


Figure 8.7: Histogram of disk service time as a load metric. The figure shows that overload is frequent in Cray Sonexion.

second (Service Level Objectives or SLO), and only 0.14% failed with timeout.

Furthermore, the operators use Kaleidoscope to characterize storage-related failures in Blue Waters. Such fine-grained characterization is not possible before the deployment of Kaleidoscope as previously deployed methods lacked joint analysis methods for identification, and disambiguation of failures.

While previous work [322] has characterized I/O failures, to the best of our knowledge, this is the first study which considers the impact of both reliability and resource-overload failures on I/O request completion time.

8.7.1 I/O Failures Caused by Reliability Failures

Kaleidoscope finds that the most common symptom of reliability failures is performance degradation that leads to I/O failures; only a very small percentage (0.057% of 346 failures (Table 8.1)) of reliability failures caused system-wide outages. For example, disk failure is tolerated by the RAID array which uses RAID resync on hot-spare disks to protect the RAID array from future failures. Such a resync or periodic scrubbing of a RAID array takes away a certain amount of bandwidth for an extended period of time, ranging from 4–12 hours, which increases completion time of I/O requests. As shown in Fig. 8.6, I/O requests during reliability failures increase the average completion time of I/O requests by up to $52.7\times$ compared to the average I/O completion time in failure-free scenarios; the 99th percentile of I/O request completion times is 31 seconds.

8.7.2 I/O Failures Caused by Resource Overloads

Kaleidoscope reveals that resource overloads frequently lead to I/O failures. We used *disk service time* (`await`), returned by `iostat`, as a metric of the load on disk devices. `await` measures the average end-to-end time for a request including device queuing and the time to service the I/O request on the disk device. `await` is different from I/O completion time, which includes the traversal time between the client and the disk.

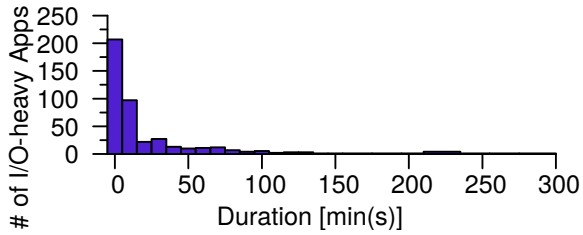


Figure 8.8: Histogram of duration of high I/O requests per application on a metadata server. The tail shows extreme I/O.

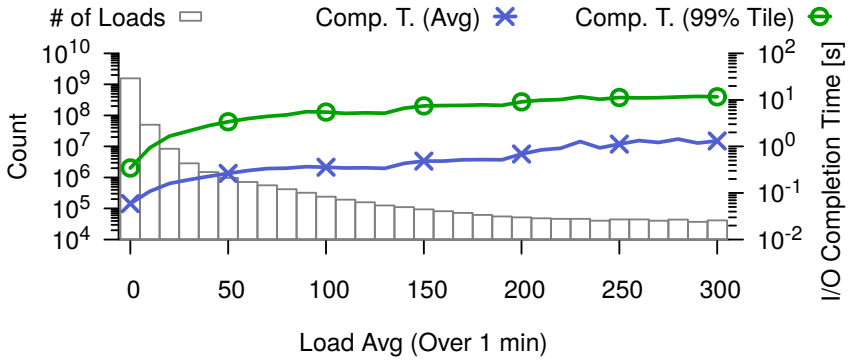


Figure 8.9: Correlation between load (i.e., `loadavg`) and latency. ("Comp. T." is the completion time of I/O requests.)

Fig. 8.7 shows a histogram of disk service time (`await`) returned by `iostat` using an event-driven measurement (triggered only when `loadavg` exceeds 50). Such anomalies occur frequently. We found 14,081 such unique events by clustering the per-disk continuous data points in time with *service times longer than one second*.

Excessive I/O. Excessive I/O requests create high load on the server and lead to disk-level contention, causing performance and stability issues. Fig. 8.8 shows a histogram of the duration of excessive I/O requests by applications to the metadata server. The duration of high I/O requests are generally small (lasting less than 10 seconds); however, there is a long tail of applications that send high I/O requests for hours. In one case, an application caused high load on the metadata server by opening and closing 75,000+ million files in 4 hours, leading to 20,000+ I/O requests per second. During that event, `loadavg` increased from 60 to 350 with the 50th and 99th percentile duration being 12 and 227 minutes, respectively.

High load. The increase of I/O request completion time has a strong correlation with the load on storage servers. High load conditions are caused by a flood of I/O requests on a storage server by either one application (e.g., extreme I/O), or multiple applications competing for resources. Fig. 8.9 shows the histogram of load across all servers. It shows

the average and 99th percentile completion time of I/O requests at different load values of the storage servers. Overall, we can see a strong correlation between an increase in load and the completion time of I/O requests. At high load (`loadavg` of 350), the average and 99th percentile I/O request completion time increases to one second and ten seconds, respectively.

8.7.3 Identifying One-off Failures

Kaleidoscope found many one-off, unique failures that do not have a common pattern and are previously unknown. Such failures can hardly be anticipated based on historical datasets. Kaleidoscope found four such failures per month on average. The following describes one of such failures.

LNET nodes serve as bridge between computing nodes and storage servers. A request from a client to an OSD (a RAID disk device) can be served by any of 4 LNET nodes. For any pair of ⟨client, OSD⟩, the group of 4 LNET nodes are fixed and chosen in round robin when routing a request. In a rare failure incident, LNET had partially failed, but the failure was not detected as Cray Sonexion uses heartbeats to detect failures. The partial failure caused LNET to drop requests passing through it, causing I/O failures. The I/O bandwidth (in MB/sec) for the applications served by the failed LNET node decreased by 25+% for multiple hours. Upon investigation, it was found that the LNET had suffered a software error that caused it to drop I/O requests for weeks. Using Kaleidoscope, we detected the failure in <5 minutes.

8.8 DISCUSSION AND LIMITATIONS

8.8.1 Interpretability of ML models

Researchers provide diverse and sometimes non-overlapping motivations for interpretability, and offer myriad notions of what attributes render models and results interpretable [363]. Below, we discuss two aspects of this general interpretability problem in the context of Kaleidoscope.

Model Interpretability. The proposed hierarchical unsupervised ML models will significantly enhance interpretability, and hence wide-spread adoption/deployment of Kaleidoscope.

1. We use models that inherently capture *all* the system modeling assumptions. For

example, Kaleidoscope through Factor Graphs (FG), a probabilistic graphical model (PGM) formalism, encodes that an I/O request failure occurs only if one or more components on the I/O request path fail.

2. Our model incorporates aspects of the storage system, i.e., topology and storage system architecture details directly into the graphical structure of the PGM. This allows the overall hierarchical model to be constructed directly from domain knowledge without requiring any pre-labeled training data (in contrast to supervised methods like deep neural networks).

Kaleidoscope can be extended to different system topologies and storage system architectures (described later in §8.8.2). Kaleidoscope automatically creates/changes the ML models with appropriate parameters using the system topology and file system I/O protocols (encoded through I/O request paths), which can be provided as an input. For example, Kaleidoscope will automatically add additional node(s) to the FG model to capture the failure state of newly added component(s) in the system. Similarly, if the I/O protocols change (i.e., the path taken by an I/O operations change), Kaleidoscope will automatically change the the factor functions to reflect new I/O paths.

Result Interpretability. System managers of Blue Waters have created several monitoring dashboards [350] to visualize live data in multiple ways. As we highlight in the §8.1, these analyses process vast amounts of telemetry data leading to cognitive overload of the system managers. Kaleidoscope strives to reduce this cognitive overload by providing intuitive charts and summaries of the telemetry data to quickly identify and understand the failure location and the failure mode (i.e., reasoning behind the ML output). An example of such a chart providing evidence of failure localization inference is shown in Fig. 8.10. The inference pointed to the existence of two concurrent failures: (i) a load issue on scratch data server 208 and (ii) an outage of projects file system metadata server. Fig. 8.10 uses a heatmap to depict a failure impact on clients (as an evidence). Each cell in the heatmap shows the ratio of operations that took longer than 1 second to the total number of operations issued during 5 minutes interval by a given client (y-axis) to each data server from Scratch, Home, and Projects Lustre partitions (x-axis), with darker color means higher ratio. Clients 0, 1, and 2 are the login nodes on Ethernet network, client 3 gives an aggregated view of all 25 Import/Export nodes on Infiniband network, and client 4 provides an aggregated view of all 64 service nodes on compute network. As seen from the figure, scratch data server 208 and project metadata server are behaving anomalously compared to rest of the cluster. Thus, Kaleidoscope, in addition to detecting and diagnosing failures, provides *significant value* in directly summarizing and visualizing the relevant evidence for a detected failure. Without Kaleidoscope the system

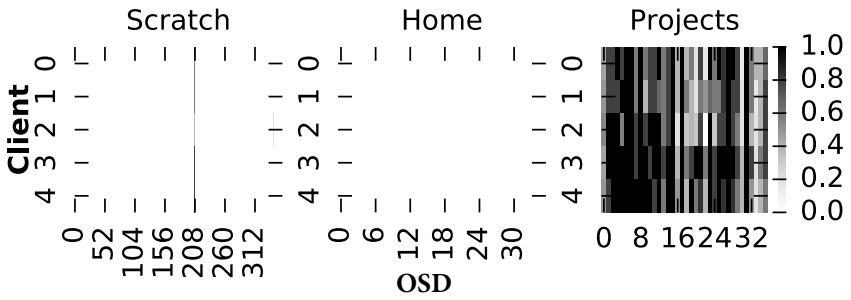


Figure 8.10: Issue on Scratch OSD 208 and Projects MDS.

managers will have to monitor a large number of failures-modes and failed-components across every instant of time.

8.8.2 Generalizing Kaleidoscope to other systems.

Kaleidoscope is not tied to a specific storage architecture. Kaleidoscope uses (i) Store Ping monitoring data for failure localization, and (ii) performance metrics and RAS logs for failure diagnosis. Performance metrics and RAS logs are already available on all storage systems. However, Store Pings must be deployed on the storage system for running Kaleidoscope. The goal of Store Ping is to test all components of the storage system such as load balancers, network, metadata servers, and object storage servers/devices (OSDs) using native storage-system operations (such as read, write, remove, etc.). Store Ping achieves this goal by pinning the files strategically (discussed in §8.4.1) onto OSDs such that the health of each of these components can be inferred. Fortunately, such support is available for all popular POSIX-compliant HPC storage systems such as Ceph [364], Gluster [365] and GPFS [366]. Let us consider Ceph. Storage cluster clients in Ceph use the CRUSH (controlled replication under scalable hashing [367]) algorithm to efficiently compute information about data location, instead of having to depend on a central lookup table (e.g., in the case of Lustre clients use MDS for file lookup). We can use CRUSH (via `crushtool`) to get the mapping rules, and use those to place the files to specific OSDs (and in doing so invoke MDS operations). Finally, recall from above that the ML models used by Kaleidoscope are not tied to specific system topology and storage protocols.

8.8.3 Dealing with large number of alarms

There is a trade-off between detecting failures quickly and generating too many alarms (due to transient failures and micro-bursts). This is a fundamental limitation of any failure detection algorithm (and it is not tied to ML). Hence, to reduce the overhead (§8.6.3), Kaleidoscope on Blue Waters is configured to collect datasets at 60s intervals. Therefore,

Kaleidoscope cannot detect micro-burst performance anomalies [368] and transient failures that are shorter than dataset collection interval (60s). Detecting transient failures/ micro-bursts is an active area of research as it allows designers to craft load-balancing and quality-of-service techniques.

In this work, we report all failures irrespective of their duration (except for the failures that are shorter than dataset collection interval and cannot be detected by Kaleidoscope). Hence, Kaleidoscope reported >26,000 failures, which is much greater than production issues. We observe more alarms than production issues because many of the I/O failure events are transient and short-lived and thus does not lead to production issues. In particular, most of the short-lived issues are related to resource overload problems (caused by one or more applications), which are filtered using heuristics discussed in §8.6.1. Moreover, as we show in §8.7, Kaleidoscope caught many failures that went unnoticed in the production for several weeks despite all the existing monitoring tools.

8.9 RELATED WORK

Kaleidoscope is built upon the wealth literature on failure detection and localization [369–375, 334, 323, 330, 333, 297, 40]. Kaleidoscope is more than a failure detector. It not only detects and localizes the failing component, but also reveals the probable causes by pinpointing the error logs or performance metrics. As discussed in §§8.1, the capability of jointly localizing and discerning the failure mode is critically important to devise the right recovery strategies. To the best of our knowledge, no existing solution provides such capability.

Kaleidoscope is the first effort for designing a hierarchical domain-driven ML-based realtime failure detection and diagnosis framework that leverages vast amounts of heterogeneous telemetry data for large-scale high-performance storage systems. Kaleidoscope's failure detection and diagnosis capabilities are fundamentally different from prior work that applies statistical or machine learning using system telemetry data: (i) prior solutions are data hungry requiring big data for training (e.g., [335, 336]), (ii) prior work supports either (a) anomaly detection [337, 338, 40], (b) failure localization [330, 331, 333, 334, 297], or (c) failure diagnosis only [339]. (iii) no prior solution handles uncertainty in telemetry data and in the system.

Kaleidoscope *proactively* detects, localizes, and diagnoses I/O timeout and slowness before the applications being affected. It uses active measurements from Store Ping monitors to support ML-based failure detection and diagnosis. It is different from passive

or reactive approaches [372, 376, 334, 40]. This requires very low monitoring overhead, i.e., Kaleidoscope has to run on a small subset of client nodes and cannot probe every single path deterministically. While active probing is a well-established technique for failure detection [323, 330], Kaleidoscope solves those key challenge by effectively modeling non-determinism and uncertainty in the distributed systems as discussed in §§8.3. As a result, Kaleidoscope reduces the number of probes by orders of magnitude compared with existing methods [323, 330, 333, 334]. In fact, active measurements are applied in limited context for storage subsystems (e.g., TOKIO [377, 322]). However, these probes have high overhead, and hence are executed once in a day.

8.10 CONCLUSION

This chapter advocates the need for identifying and diagnosing resource overload and reliability failures jointly to effectively coordinate recovery strategy. We build Kaleidoscope and deploy it on a petascale production system to disambiguate component failures from resource overload/contention issues.

CHAPTER 9: CONCLUSION

Society as a whole is going to witness exponential growth in the use of data-driven automation techniques in critical application domains such as scientific and cloud computing, healthcare, transportation, agriculture, and manufacturing. The widespread adoption of such systems in a human-centric environments necessitates our understanding of their decision-making processes in the presence of a wide range of uncertainties, from specification to real-time operations. These next-generation data-driven systems, especially ML/AI-driven systems, demand an ever-increasing level of system dependability not available today. The classical approach to dependability is based upon component reliability views and fault/error/attack management at the architectural level. While necessary, the classical approaches are not sufficient: novel methods must be developed to account for autonomy and safety requirements.

This thesis is a step in that direction. We develop novel causality-driven assessment techniques that meet those demands and provide the theory and foundation for designing dependable data-driven systems. Methods proposed in this work will allow designers to assess and ensure the dependability of such systems.

We then showcase these techniques on two complex, mission-critical, data-driven systems: (i) autonomous vehicles (particularly, self-driving cars) and (ii) large-scale computing infrastructures (HPC and Cloud). Furthermore, the techniques proposed in this work will pave the way to ensuring the dependability and performance of other autonomous systems, such as unmanned aerial vehicles, agricultural robots, and kitchen bots, among others.

APPENDIX A: OTHER WORK

In this section, I briefly describe other research work that was conducted in parallel to this thesis.

A.1 AUTONOMOUS VEHICLES

Scalable Fuzzing of Driving Scenarios for AV Testing. We propose AV-FUZZER [378], a testing framework, to find the safety violations of an autonomous vehicle (AV) in the presence of an evolving traffic environment. We perturb the driving maneuvers of traffic participants to create situations in which an AV can run into safety violations. To optimally search for the perturbations to be introduced, we leverage domain knowledge of vehicle dynamics and a genetic algorithm to minimize the safety potential of an AV over its projected trajectory. The values of the perturbation determined by this process provide parameters that define the participants' trajectories. To improve the efficiency of the search, we design a local fuzzer that increases the exploitation of local optima in the areas where highly likely safety- hazardous situations are observed. By repeating the optimization with significantly different starting points in the search space, AV-FUZZER determines several diverse AV safety violations. We demonstrate AV-FUZZER on an industrial-grade AV platform, Baidu Apollo, and find five distinct types of safety violations in a short period of time. In comparison, other existing techniques can find at most two. We analyze the safety violations found in Apollo and discuss their overarching causes.

A.2 HIGH-PERFORMANCE COMPUTING (HPC) AND CLOUD

Characterizing Application Job Failures in HPC. Node downtime and failed jobs in a computing cluster translate into wasted resources and user dissatisfaction. Therefore, understanding why nodes and jobs fail in HPC clusters is essential. In this work [45], we provide analyses of node and job failures in two university-wide computing clusters at two Tier I US research universities. We analyzed approximately 3.0M job execution data of System A and 2.2M of System B with data sources coming from accounting logs, resource usage for all primary local and remote resources (memory, IO, network), and node failure data. We observe different kinds of correlations of failures with resource usage and propose a job failure prediction model to trigger event-driven checkpointing and

avoid wasted work. Additionally, we present user history based resource usage and runtime prediction models. These models have the potential to avoid system-related issues, such as contention, and to improve quality of service, such as lower mean queue time, if their predictions are used to make a more informed scheduling decision. As a proof of concept, we simulate an easy backfill scheduler to use predictions of one of these models, i.e., runtime, and show the improvements in terms of lower mean queue time. Arising out of these observations, we provide generalizable insights for cluster management to improve reliability, such as, for some execution environments local contention dominates, while for others system-wide contention dominates.

Application-oriented Reliability Models for HPC. Reliability analysis and performance evaluation are complementary methods to quantify nonfunctional aspects of a system. However, a range of factors such as concurrency and heterogeneity quickly exacerbate the state-space explosion problem when attempting detailed system-level modeling and simulation of HPC systems. To overcome these impediments to modeling and analysis, our work [379] develops a hierarchical model of an application that implements checkpointing running in an HPC environment subject to application, network, and system-wide outages. The modeling approach ensures that the number of states is linear in the number of checkpoints and possesses a low constant factor for the number of recovery states most relevant to the external influences contributing to degraded application performance. We illustrate the types of analysis enabled by the model through a series of examples, with parameters determined empirically from data logs of the Blue Waters supercomputer located at the University of Illinois at Urbana–Champaign. A comprehensive comparative analysis of the model parameters indicates that lowering the failure rate of network nodes would most significantly reduce application downtime. We also discuss how the modeling approach can be used to objectively assess both current and hypothetical future systems to identify competitive designs and enhancements.

Fault Injection-driven HPC Assessment. In this work [56], we present a set of fault injection experiments performed on the ACES (LANL/SNL) Cray XE supercomputer Cielo. We use this experimental campaign to improve the understanding of failure causes and propagation that we observed in the field failure data analysis of NCSA's Blue Waters. We use the data collected from the logs and from network performance counter data (i) to characterize the fault-error-failure sequence and recovery mechanisms in the Gemini network and in the Cray compute nodes, (ii) to understand the impact of failures on the system and the user applications at different scale, and (iii) to identify and recreate fault scenarios that induce unrecoverable failures, in order to create new tests for system and application design. The faults were injected through special input commands to bring

down network links, directional connections, nodes, and blades. We present the extensions that will be needed to apply our methodologies of injection and analysis to the Cray XC (Aries) systems.

Congestion Mitigation. Modern HPC systems concurrently execute multiple distributed applications that contend for the high-speed network leading to congestion. Consequently, application runtime variability and suboptimal system utilization are observed in production systems. To address these problems, we propose Netscope [42], a congestion mitigation framework based on a novel delay sensitivity metric that quantifies the impact of congestion on application runtime. Netscope uses delay sensitivity estimates to drive a congestion mitigation mechanism to selectively throttle applications that are less susceptible to congestion. We evaluate Netscope on two Cray Aries systems, including a production super- computer, on common scientific applications. Our evaluation shows that Netscope has a low training cost and accurately estimates the impact of congestion on application runtime with a correlation between 0.7 and 0.9. Moreover, Netscope reduces application tail runtime increase by up to 16.3×while improving the median system utility by 12%

Mitigating SLO Violations in Microservices. User-facing latency-sensitive web services include numerous distributed, intercommunicating microservices that promise to simplify software development and operation. However, multiplexing of compute resources across microservices is still challenging in production because contention for shared resources can cause latency spikes that violate the service- level objectives (SLOs) of user requests. FIRM [47] is an intelligent fine-grained resource management framework for predictable sharing of resources across microservices to drive up overall utilization. FIRM leverages online telemetry data and machine-learning methods to adaptively (a) detect/localize microservices that cause SLO violations, (b) identify low-level resources in contention, and (c) take actions to mitigate SLO violations via dynamic reprovisioning. Experiments across four microservice benchmarks demon- strate that FIRM reduces SLO violations by up to 16×while reducing the overall requested CPU limit by up to 62%. More- over, FIRM improves performance predictability by reducing tail latencies by up to 11×.

REFERENCES

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

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

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

[4] J. Peters, "Prolonged AWS outage takes down a big chunk of the internet," https://www.theverge.com/2020/11/25/21719396/amazon-web-services-aws-outage-down-internet, Accessed 13 September 2021.

[5] S. Moss, "How California's wildfires took down a supercomputer," https://www.datacenterdynamics.com/en/analysis/how-californias-wildfires-took-down-supercomputer/, accessed 13 September 2021.

[6] S. S. Banerjee, S. Jha, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 586–597.

[7] J. Lu, H. Sibai, and E. Fabry, "Adversarial examples that fool detectors," *arXiv preprint arXiv:1712.02494*, 2017.

[8] S. Jha, S. S. Banerjee, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "AVFI: Fault injection for autonomous vehicles," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2018, pp. 55–56.

[9] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[10] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[11] D. Kang, D. Raghavan, P. Bailis, and M. Zaharia, "Model assertions for monitoring and improving ml models," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020. [Online]. Available: https://proceedings.mlsys.org/paper/2020/file/a2557a7b2e94197ff767970b67041697-Paper.pdf pp. 481–496.

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

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

[14] Nvidia, "Nvidia Drive," https://developer.nvidia.com/driveworks.

[15] "Apollo Open Platform," http://apollo.auto, accessed: 2018-09-02.

[16] "Openpilot git repo." [Online]. Available: https://github.com/commaai/openpilot

[17] K. Chung, Z. T. Kalbarczyk, and R. K. Iyer, "Availability attacks on computing systems through alteration of environmental control: Smart malware approach," in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302509.3311041 p. 1–12.

[18] F. Mujica, "Scalable electronics driving autonomous vehicle technologies," *Texas Instruments White Paper*, 2014.

[19] A. Abdulkhaleq, D. Lammering, S. Wagner, J. Röder, N. Balbierer, L. Ramsauer, T. Raste, and H. Boehmert, "A systematic approach based on STPA for developing a dependable architecture for fully automated driving vehicles," *Procedia Engineering*, vol. 179, pp. 41–51, 2017, 4th European STAMP Workshop 2016, ESW 2016, 13-15 September 2016, Zurich, Switzerland. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877705817312109

[20] Waymo, "On the Road to Fully Self-Driving," Waymo Safety Report https://assets.documentcloud.org/documents/4107762/Waymo-Safety-Report-2017.pdf, accessed: 2017-11-27.

[21] N. H. Amer, H. Zamzuri, K. Hudha, and Z. A. Kadir, "Modelling and control strategies in path tracking control for autonomous ground vehicles: A review of state of the art and challenges," *Journal of Intelligent & Robotic Systems*, vol. 86, no. 2, pp. 225–254, 2017. [Online]. Available: https://doi.org/10.1007/s10846-016-0442-0

[22] 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)*, 2012, pp. 3354–3361.

[23] California Department of Motor Vehicles, "Testing of autonomous vehicles," https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/testing, accessed: 2017-11-27.

[24] 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 *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2019, pp. 112–124.

[25] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2723372.2723711 p. 331–346.

[26] 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 *ICLR 2020 : Eighth International Conference on Learning Representations"*, 2020.

[27] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (Dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[28] J. Evans, "The post-exponential era of AI and Moore's law." https://techcrunch.com/2019/11/10/the-post-exponential-era-of-ai-and-moores-law/, 2019, accessed September 14, 2021,.

[29] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 2016, pp. 50–56.

[30] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098843 pp. 197–210.

[31] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *SIGPLAN Not.*, vol. 48, no. 4, pp. 77–88, Mar. 2013.

[32] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 127–144.

[33] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019. [Online]. Available: http://proceedings.mlr.press/v97/jay19a.html pp. 3050–3059.

[34] A. Dziedzic, V. Sathya, M. I. Rochman, M. Ghosh, and S. Krishnan, "Machine learning enabled spectrum sharing in dense lte-u/wi-fi coexistence scenarios," *CoRR*, vol. abs/2003.13652, 2020. [Online]. Available: https://arxiv.org/abs/2003.13652

[35] Y. Kong, H. Zang, and X. Ma, "Improving tcp congestion control with machine intelligence," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, ser. NetAI'18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3229543.3229550 p. 60–66.

[36] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 610–621.

[37] C. Di Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 25–36.

[38] S. Jha, V. Formicola, C. Di Martino, M. Dalton, W. T. Kramer, Z. Kalbarczyk, and R. K. Iyer, "Resiliency of HPC interconnects: A case study of interconnect failures and recovery in Blue Waters," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 915–930, 2017.

[39] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li, "Fail-slow at scale: Evidence of hardware performance faults in large production systems," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, Oakland, CA, USA, FEB 2018.

[40] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, jul 2019. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/panda pp. 47–62.

[41] R. Iyer, L. Young, and P. Iyer, "Automatic recognition of intermittent failures: an experimental study of field data," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 525–537, 1990.

[42] A. Patke, S. Jha, H. Qiu, J. Brandt, A. Gentile, J. Greenseid, Z. Kalbarczyk, and R. K. Iyer, "Delay sensitivity-driven congestion mitigation for hpc systems," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3447818.3460362 p. 342–353.

[43] S. Jha, A. Patke, J. Brandt, A. Gentile, M. Showerman, E. Roman, Z. T. Kalbarczyk, B. Kramer, and R. K. Iyer, "A study of network congestion in two supercomputing high-speed interconnects," in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2019, pp. 45–48.

[44] S. Jha, A. Patke, J. Brandt, A. Gentile, B. Lim, M. Showerman, G. Bauer, L. Kaplan, Z. Kalbarczyk, W. Kramer, and R. Iyer, "Measuring congestion in high-performance datacenter interconnects," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, feb 2020. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/jha pp. 37–57.

[45] R. Kumar, S. Jha, A. Mahgoub, R. Kalyanam, S. Harrell, X. C. Song, Z. Kalbarczyk, W. Kramer, R. Iyer, and S. Bagchi, "The mystery of the failing jobs: Insights from operational data from two university-wide computing systems," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 158–171.

[46] R. Ganesan, S. Sarkar, G. Goel, and C. D. Martino, "Measurements-based analysis of workload-error relationship in a production saas cloud," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, 2012, pp. 96–105.

[47] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/qiu pp. 805–825.

[48] 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.

[49] B. Bode, M. Butler, T. Dunning, T. Hoefler, W. Kramer, W. Gropp, and W.-m. Hwu, "The Blue Waters super-system for super-science," in *Contemporary high performance computing*. Chapman and Hall/CRC, 2013, pp. 339–366.

[50] "Blue Waters," https://bluewaters.ncsa.illinois.edu.

[51] W. Kramer, M. Butler, G. Bauer, K. Chadalavada, and C. Mendes, "Blue Waters parallel I/O storage sub-system," *High Performance Parallel I/O*, pp. 17–32, 2015.

[52] 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*, ser. SIGMOD '00. New York, NY, USA: Association for Computing Machinery, 2000. [Online]. Available: https://doi.org/10.1145/342009.335388 p. 93–104.

[53] "Platform for scalable testing of autonomous vehicle safety," https://www.sciencedaily.com/releases/2019/10/191025170813.htm.

[54] "Researchers look to increase safety in self-driving cars," https://dailyillini.com/news/2019/11/07/researchers-look-to-increase-safety-in-self-driving-cars/.

[55] "Self-driving thunder! U.S. experts found 561 faults in Baidu Apollo and Nvidia DriveAV in 4 hours," https://www.guancha.cn/industry-science/2019_11_01_523593.shtml.

[56] V. Formicola, S. Jha, D. Chen, F. Deng, A. Bonnie, M. Mason, J. Brandt, A. Gentile, L. Kaplan, J. Repik et al., "Understanding fault scenarios and impacts through fault injection experiments in cielo," *arXiv preprint arXiv:1907.01019*, 2019.

[57] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "{FIRM}: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 805–825.

[58] M. Gerla, E. K. Lee, G. Pau, and U. Lee, "Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Mar 2014, pp. 241–246.

[59] U. Ozguner, C. Stiller, and K. Redmill, "Systems for safety and autonomous behavior in cars: The DARPA grand challenge experience," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 397–412, 2007.

[60] M. Buehler, K. Iagnemma, and S. Singh, Eds., *The DARPA Urban Challenge*. Springer Berlin Heidelberg, 2009. [Online]. Available: https://doi.org/10.1007/978-3-642-03991-1

[61] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. M. Howard, S. Kolski, A. Kelly, M. Likhachev, M. McNaughton, N. Miller, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, B. Salesky, Y.-W. Seo, S. Singh, J. Snider, A. Stentz, W. R. Whittaker, Z. Wolkowicki, J. Ziglar, H. Bae, T. Brown, D. Demitrish, B. Litkouhi, J. Nickolaou, V. Sadekar, W. Zhang, J. Struble, M. Taylor, M. Darms, and D. Ferguson, "Autonomous driving in urban environments: Boss and the urban challenge," *J. Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[62] A. Chatham, "Google's self-driving cars: The technology, capabilities, and challenges," in *2013 Embedded Linux Conf., Feb*, 2013, pp. 20–24.

[63] C. Urmson, "Realizing self-driving vehicles," in *2012 IEEE Intelligent Vehicles Symposium (IV). Alcalá des Henares, Spanien*, 2012.

[64] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, 2016.

[65] W. Payre, J. Cestac, and P. Delhomme, "Intention to use a fully automated car: Attitudes and a priori acceptability," *Transportation Research Part F: Traffic Psychology and Behaviour*, vol. 27, pp. 252–263, Nov 2014.

[66] S. Shladover, D. Su, and X.-Y. Lu, "Impacts of cooperative adaptive cruise control on freeway traffic flow," *Transportation Research Record: J. the Transportation Research Board*, vol. 2324, pp. 63–70, Dec 2012.

[67] G. E. Marchant and R. A. Lindor, "The coming collision between autonomous vehicles and the liability system," *Santa Clara L. Rev.*, vol. 52, p. 1321, 2012.

[68] M. Parent et al., "Legal issues and certification of the fully automated vehicles: Best practices and lessons learned," *CityMobil2 Rep.*, 2013.

[69] J. M. Anderson, N. Kalra, K. D. Stanley, P. Sorensen, C. Samaras, and T. A. Oluwatola, *Autonomous Vehicle Technology: A Guide for Policymakers*. Santa Monica, CA: RAND Corporation, 2016.

[70] D. J. Fagnant and K. Kockelman, "Preparing a nation for autonomous vehicles: Opportunities, barriers and policy recommendations," *Transportation Research Part A: Policy and Practice*, vol. 77, pp. 167–181, Jul 2015.

[71] L. Fraade-Blanar and N. Kalra, "Autonomous vehicles and federal safety standards: An exemption to the rule?" RAND Corp., Tech. Rep. PE-258-RC, 2017.

[72] D. G. Groves and N. Kalra, "Enemy of good," RAND Corp., Tech. Rep. RR-2150-RC, 2017.

[73] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[74] S. Petti and T. Fraichard, "Safe motion planning in dynamic environments," in *2005 IEEE/RSJ International Conf. Intelligent Robots and Systems*, Aug 2005, pp. 2210–2215.

[75] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126964

[76] NVIDIA, "Introducing Xavier, the Nvidia AI supercomputer for the future of autonomous transportation," https://blogs.nvidia.com/blog/2016/09/28/xavier, accessed: 2017-11-27.

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

[78] N. Leveson, *Engineering a safer world: Systems thinking applied to safety*. MIT press, 2011.

[79] California Department of Motor Vehicles, "Deployment of autonomous vehicles for public operation," https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/auto, accessed: 2017-11-27.

[80] F. M. Favarò, N. Nader, S. O. Eurich, M. Tripp, and N. Varadaraju, "Examining accident reports involving autonomous vehicles in California," *PLoS one*, vol. 12, no. 9, p. e0184952, 2017.

[81] F. Favarò, S. Eurich, and N. Nader, "Autonomous vehicles' disengagements: Trends, triggers, and regulatory limitations," *Accident Analysis & Prevention*, vol. 110, pp. 136–148, 2018.

[82] R. Smith, "An overview of the Tesseract OCR Engine," in *Ninth International Conf. Document Analysis and Recognition*, vol. 2, Sep 2007, pp. 629–633.

[83] H. Alemzadeh, "Data-driven resiliency assessment of medical cyber-physical systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2016.

[84] V. V. Dixit, S. Chand, and D. J. Nair, "Autonomous vehicles: Disengagements, accidents and reaction times," *PLOS ONE*, vol. 11, no. 12, pp. 1–14, 12 2016. [Online]. Available: https://doi.org/10.1371/journal.pone.0168054

[85] D. B. Fambro, *Determination of stopping sight distances (Report / National Cooperative Highway Research Program)*. National Academy Press, 1997.

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

[87] National Highway Traffic Safety Administration (NHTSA), "2015 motor vehicle crashes: overview DOT HS 812 318," *Traffic Safety Facts Research Note*, pp. 1–9, 2016.

[88] Federal Highway Administration (FHWA), "Traffic Volume Trends." https://www.fhwa.dot.gov/policyinformation/travel_monitoring/tvt.cfm, accessed: 2017-11-27.

[89] US DOT FAA, "System design and aalysis," Tech. Rep. AC 25.1309-1A, Jun 1988.

[90] H. Alemzadeh, J. Raman, N. Leveson, Z. Kalbarczyk, and R. K. Iyer, "Adverse events in robotic surgery: A retrospective study of 14 years of FDA data," *PLOS ONE*, vol. 11, no. 4, pp. 1–20, 04 2016. [Online]. Available: https://doi.org/10.1371/journal.pone.0151470

[91] National Transportation Safety Board, "Aviation statistics: Review of accident data," https://www.ntsb.gov/investigations/data/Pages/aviation_stats.aspx, accessed: 2017-11-27.

[92] H. Alemzadeh, R. K. Iyer, Z. Kalbarczyk, and J. Raman, "Analysis of safety-critical computer failures in medical devices," *IEEE Security Privacy*, vol. 11, no. 4, pp. 14–26, 2013.

[93] U.S. Department of Transportation and Federal Highway Administration and Office of Highway Policy Information and National Household Travel Survey., "Our Nation's Highways: 2008," https://www.fhwa.dot.gov/policyinformation/pubs/pl08021/index.cfm, accessed: 2017-11-27.

[94] P. Plötz, N. Jakobsson, and F. Sprei, "On the distribution of individual daily driving distances," *Transportation Research Part B: Methodological*, vol. 101, pp. 213–227, 2017.

[95] G. Stanek, D. Langer, B. Müller-Bessler, and B. Huhnke, "Junior 3: A test platform for advanced driver assistance systems," in *Intelligent Vehicles Symposium (IV), 2010 IEEE*, 2010, pp. 143–149.

[96] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling, and S. Thrun, "Towards fully autonomous driving: Systems and algorithms," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, Jun 2011, pp. 163–168.

[97] "Road vehicles: Functional safety," International Organization for Standardization, Geneva, CH, Standard, nov 2011.

[98] S. M. Casner, E. L. Hutchins, and D. Norman, "The challenges of partially automated driving," *Comm. of the ACM*, vol. 59, no. 5, pp. 70–77, Apr 2016.

[99] A. Reschka, "Safety concept for autonomous vehicles," in *Autonomous Driving: Technical, Legal and Social Aspects*, M. Maurer, J. C. Gerdes, B. Lenz, and H. Winner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 473–496.

[100] P. Koopman and M. Wagner, "Autonomous vehicle safety: An interdisciplinary challenge," *IEEE Intelligent Transportation Systems Magazine*, vol. 9, no. 1, pp. 90–96, Spring 2017.

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

[102] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "DryVR: Data-driven verification and compositional reasoning for automotive systems," in *Computer Aided Verification*. Springer International Publishing, 2017, pp. 441–461.

[103] SAE International, *Cybersecurity Guidebook for Cyber-Physical Vehicle Systems*, Jan 2016.

[104] J. Joy and M. Gerla, "Privacy risks in vehicle grids and autonomous cars," in *Proceedings of the 2nd ACM International Workshop on Smart, Autonomous, and Connected Vehicular Systems and Services*, ser. CarSys '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3131944.3133938 p. 19–23.

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

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

[107] J. R. Bitner, J. Jain, M. S. Abadir, J. A. Abraham, and D. S. Fussell, "Efficient algorithmic circuit verification using indexed BDDs," in *Digest of Papers: 24th Symposium on Fault-Tolerant Computing*, 1994. [Online]. Available: https://doi.org/10.1109/FTCS.1994.315633 pp. 266–275.

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

[109] R. K. Roy, T. M. Niermann, J. H. Patel, J. A. Abraham, and R. A. Saleh, "Compaction of ATPG-generated test sequences for sequential circuits," in *Digest of Technical Papers, 1988 IEEE International Conference on Computer-Aided Design*, 1988. [Online]. Available: https://doi.org/10.1109/ICCAD.1988.122533 pp. 382–385.

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

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

[112] L. Fraade-Blanar, M. S. Blumenthal, J. M. Anderson, and N. Kalra, *Measuring Automated Vehicle Safety: Forging a Framework*. Santa Monica, CA: RAND Corporation, 2018.

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

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

[115] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

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

[117] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs." in *OSDI*, vol. 8, 2008, pp. 267–280.

[118] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 249–258.

[119] NHTSA, "Automated driving systems: A vision for safety," https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13069a-ads2.0_090617_v9a_tag.pdf, 2017.

[120] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *Proc. of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[121] B. Salami, O. Unsal, and A. Cristal, "On the resilience of RTL NN accelerators: Fault characterization and mitigation," *arXiv preprint arXiv:1806.09679*, 2018.

[122] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 17.

[123] A. H. M. Rubaiyat, Y. Qin, and H. Alemzadeh, "Experimental resilience assessment of an open-source driving agent," in *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2018, pp. 54–63.

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

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

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

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

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

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

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

[131] D. Koller, J. Weber, T. Huang, J. Malik, G. Ogasawara, B. Rao, and S. Russell, "Towards robust automatic traffic scene analysis in real-time," in *Pattern Recognition, 1994. Vol. 1-Conference A: Computer Vision & Image Processing., Proceedings of the 12th IAPR International Conference on*, vol. 1. IEEE, 1994, pp. 126–131.

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

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

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

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

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

[137] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.

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

[139] A. Houenou, P. Bonnifait, V. Cherfaoui, and W. Yao, "Vehicle trajectory prediction based on motion model and maneuver recognition," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 4363–4369.

[140] 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*, ser. Proceedings of Machine Learning Research, S. Levine, V. Vanhoucke, and K. Goldberg, Eds., vol. 78. PMLR, 13–15 Nov 2017. [Online]. Available: http://proceedings.mlr.press/v78/dosovitskiy17a.html pp. 1–16.

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

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

[143] NEOUSYS, "Nuvo-6108GC GPU computing platform Nvidia RTX 2080-GTX 1080TI-Titanx," https://www.neousys-tech.com/en/product/application/gpu-computing/nuvo-6108gc-gpu-computing, accessed: 2018-11-28.

[144] S. Jha, T. Tsai, S. Hari, M. Sullivan, Z. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Kayotee: A fault injection-based system to assess the safety and reliability of autonomous vehicles to faults and errors," in *Third IEEE International Workshop on Automotive Reliability & Test*. IEEE, 2018.

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

[146] J. Lu, H. Sibai, E. Fabry, and D. Forsyth, "No need to worry about adversarial examples in object detection in autonomous vehicles," *arXiv preprint arXiv:1707.03501*, 2017.

[147] K. Pei, Y. Cao, J. Yang, and S. Jana, "Towards practical verification of machine learning: The case of computer vision systems," *arXiv preprint arXiv:1712.01785*, 2017.

[148] H. Lakkaraju, E. Kamar, R. Caruana, and E. Horvitz, "Identifying unknown unknowns in the open world: Representations and policies for guided exploration." in *AAAI*, vol. 1, 2017, p. 2.

[149] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial sensor attack on LiDAR-based perception in autonomous driving," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3319535.3339815 p. 2267–2281.

[150] A. Boloor, K. Garimella, X. He, C. Gill, Y. Vorobeychik, and X. Zhang, "Attacking vision-based perception in end-to-end autonomous driving models," *Journal of Systems Architecture*, vol. 110, p. 101766, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762120300606

[151] 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 visual classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[152] LG Electronics, "LGSVL simulator," https://www.lgsvlsimulator.com/, 2018.

[153] W. Luo, X. Zhao, and T.-K. Kim, "Multiple object tracking: A review," *CoRR*, vol. abs/1409.7618, 2014. [Online]. Available: http://arxiv.org/abs/1409.7618

[154] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[155] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[156] G. Welch, G. Bishop et al., *An introduction to the Kalman filter*. Los Angeles, CA, 1995.

[157] C. Huang, B. Wu, and R. Nevatia, "Robust object tracking by hierarchical association of detection responses," in *European Conference on Computer Vision*. Springer, 2008, pp. 788–801.

[158] K.-T. Cho and K. G. Shin, "Fingerprinting electronic control units for vehicle intrusion detection," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, august 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/cho pp. 911–927.

[159] 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 *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 447–462.

[160] D. Rezvani, "Hacking automotive ethernet cameras," https://argus-sec.com/hacking-automotive-ethernet-cameras/, 2018.

[161] "IEEE 802.1: 802.1BA - Audio video bridging (AVB) systems," http://www.ieee802.org/1/pages/802.1ba.html, (Accessed on 12/12/2019).

[162] 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.

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

[164] O. Pfeiffer, "Implementing scalable CAN security with CAN crypt," *Embedded Systems Academy*, 2017.

[165] K. Manandhar, X. Cao, F. Hu, and Y. Liu, "Detection of faults and attacks including false data injection attack in smart grid using Kalman filter," *IEEE transactions on control of network systems*, vol. 1, no. 4, pp. 370–379, 2014.

[166] "Apollo," https://github.com/ApolloAuto/apollo.

[167] LG Electronics, "Modified Apollo 5.0," https://github.com/lgsvl/apollo-5.0.

[168] "Unity Engine," http://unity.com.

[169] J. Gregory, *Game engine architecture*. AK Peters/CRC Press, 2017.

[170] "Sensor configuration example," https://www.lgsvlsimulator.com/docs/apollo5-0-json-example/.

[171] I. A. Sumra, I. Ahmad, H. Hasbullah et al., "Classes of attacks in VANET," in *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*. IEEE, 2011, pp. 1–5.

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

[173] E. Yağdereli, C. Gemci, and A. Z. Aktaş, "A study on cyber-security of autonomous and unmanned vehicles," *The Journal of Defense Modeling and Simulation*, vol. 12, no. 4, pp. 369–381, 2015.

[174] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[175] R. Gallager, "Low-density parity-check codes," *IRE Transactions on information theory*, vol. 8, no. 1, pp. 21–28, 1962.

[176] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627–641, 1999.

[177] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors - A survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, 1988.

[178] L. A. Clarke and D. S. Rosenblum, "A historical perspective on runtime assertion checking in software development," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, p. 25–37, may 2006. [Online]. Available: https://doi.org/10.1145/1127878.1127900

[179] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.

[180] 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. [Online]. Available: https://doi.org/10.1145/3323917

[181] 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.

[182] 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.

[183] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, 2000, pp. 25–36.

[184] 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.

[185] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, 2003, pp. 98–109.

[186] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, 2007, pp. 317–326.

[187] H. Jeon and M. Annavaram, "Warped-DMR: Light-weight error detection for GPGPU," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 37–47.

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

[189] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1491–1501, 1985.

[190] P. Ammann and J. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 418–425, 1988.

[191] W. W. Peterson, W. Peterson, E. J. Weldon, and E. J. Weldon, *Error-Correcting Codes, second edition*. MIT press, 1972.

[192] D. Chen, B. Zhou, V. Koltun, and P. Krähenbühl, "Learning by cheating," in *Proceedings of the Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, L. P. Kaelbling, D. Kragic, and K. Sugiura, Eds., vol. 100. PMLR, 30 Oct–01 Nov 2020. [Online]. Available: http://proceedings.mlr.press/v100/chen20a.html pp. 66–75.

[193] 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," 2016.

[194] R. Moore-Colyer, "Nvidia Xavier Supercomputer aims to turn cars into AI on wheels."

[195] S. Mitra, N. Saxena, and E. McCluskey, "A design diversity metric and reliability analysis for redundant systems," in *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, 1999, pp. 662–671.

[196] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The KITTI dataset," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, 2013. [Online]. Available: https://doi.org/10.1177/0278364913491297

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

[198] 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*, 2014, pp. 375–382.

[199] "PinFI," https://github.com/DependableSystemsLab/pinfi.

[200] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, "NVBitFI: Dynamic fault injection for GPUs," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 284–291.

[201] Z. Chen, G. Li, and K. Pattabiraman, "A low-cost fault corrector for deep neural networks through range restriction," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 1–13.

[202] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, "Robust anomaly detection for multivariate time series through stochastic recurrent neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3292500.3330672 p. 2828–2837.

[203] H. Zhao, S. K. S. Hari, T. Tsai, M. B. Sullivan, S. W. Keckler, and J. Zhao, "Suraksha: A quantitative AV safety evaluation framework to analyze safety implications of perception design choices," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2021, pp. 35–38.

[204] "Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices (JESD89A)," October 2006.

[205] G. L. Hicks, L. D. Howe Jr, and F. A. Zurla Jr, "Instruction retry mechanism for a data processing system," aug 1977, uS Patent 4,044,337.

[206] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for GPU error detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 842–853.

[207] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for GPGPUs," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 287–300.

[208] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Microelectronics division*, vol. 11, pp. 1–23, 1997.

[209] J. E. Barth Jr, C. E. Drake, J. A. Fifield, W. P. Hovis, H. L. Kalter, S. C. Lewis, D. J. Nickel, C. H. Stapper, and J. A. Yankosky, "Dynamic RAM with on-chip ECC and optimized bit and word redundancy," jul 1992, uS Patent 5,134,616.

[210] M. B. Sullivan, S. K. S. Hari, B. Zimmer, T. Tsai, and S. W. Keckler, "SwapCodes: Error codes for hardware-software cooperative GPU pipeline error detection," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 762–774.

[211] R. Nathan and D. J. Sorin, "Argus-G: Comprehensive, low-cost error detection for GPGPU cores," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 13–16, 2015.

[212] B. H. Meyer, B. H. Calhoun, J. Lach, and K. Skadron, "Cost-effective safety and fault localization using distributed temporal redundancy," in *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 125–134.

[213] J. Fu, Q. Yang, R. Poss, C. R. Jesshope, and C. Zhang, "On-demand thread-level fault detection in a concurrent programming environment," in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2013, pp. 255–262.

[214] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux et al., "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 331–342.

[215] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[216] S. K. S. Hari, M. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2021.

[217] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *2011 Design, Automation & Test in Europe*. IEEE, 2011, pp. 1–6.

[218] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, "Anomaly detection using autoencoders in high performance computing systems," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 9428–9433, Jul. 2019. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/4993

[219] F. Haas, S. Weis, T. Ungerer, G. Pokam, and Y. Wu, "Fault-tolerant execution on COTS multi-core processors with hardware transactional memory support," in *Architecture of Computing Systems - ARCS 2017*, J. Knoop, W. Karl, M. Schulz, K. Inoue, and T. Pionteck, Eds. Cham: Springer International Publishing, 2017, pp. 16–30.

[220] M. S. Alhakeem, P. Munk, R. Lisicki, H. Parzyjegla, H. Parzyjegla, and G. Muehl, "A framework for adaptive software-based reliability in COTS many-core processors," in *ARCS 2015 - The 28th International Conference on Architecture of Computing Systems. Proceedings*, 2015, pp. 1–4.

[221] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, p. 30–39, december 2010. [Online]. Available: https://doi.org/10.1145/1899928.1899932

[222] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[223] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Application-based metrics for strategic placement of detectors," in *11th Pacific Rim International Symposium on Dependable Computing (PRDC'05)*, 2005, pp. 8 pp.–.

[224] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Symplfied: Symbolic program-level fault injection and error detection framework," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 472–481.

[225] 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.

[226] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "PLR: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, 2009.

[227] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 240–251.

[228] Person and D. Shepardson, "U.s. traffic deaths up during pandemic even though mileage down -data," Sep 2021. [Online]. Available: https://www.reuters.com/world/us/us-traffic-deaths-jump-105-early-2021-2021-09-02/

[229] R. E. Barlow and F. Proschan, "Importance of system components and fault tree events," *Stochastic Processes and their applications*, vol. 3, no. 2, pp. 153–173, 1975.

[230] S. M. LaValle et al., "Rapidly-exploring random trees: A new tool for path planning," 1998.

[231] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuscenes: A multimodal dataset for autonomous driving," *arXiv preprint arXiv:1903.11027*, 2019.

[232] N. David, L. Hon-Leung, N. Julia, and W. Yizhou, "An introduction to the safety force field," https://www.nvidia.com/content/dam/en-zz/Solutions/self-driving-cars/safety-force-field/an-introduction-to-the-safety-force-field-updated.pdf, 2019.

[233] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "On a formal model of safe and scalable self-driving cars," *arXiv preprint arXiv:1708.06374*, 2017.

[234] A. Filos, P. Tigkas, R. Mcallister, N. Rhinehart, S. Levine, and Y. Gal, "Can autonomous vehicles identify, recover from, and adapt to distribution shifts?" in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020. [Online]. Available: https://proceedings.mlr.press/v119/filos20a.html pp. 3145–3153.

[235] J. Philion, A. Kar, and S. Fidler, "Learning to evaluate perception models using planner-centric metrics," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[236] J. T. Schwartz and M. Sharir, "On the "piano movers" problem. ii. general techniques for computing topological properties of real algebraic manifolds," *Advances in Applied Mathematics*, vol. 4, no. 3, pp. 298–351, 1983. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0196885883900143

[237] J. Canny, *The complexity of robot motion planning*. MIT press, 1988.

[238] D. J. C. Mackay, "Probable networks and plausible predictions — a review of practical bayesian methods for supervised neural networks," vol. 6, no. 3, pp. 469–505, jan 1995. [Online]. Available: https://doi.org/10.1088/0954-898x_6_3_011

[239] I. Gog, S. Kalra, P. Schafhalter, M. A. Wright, J. E. Gonzalez, and I. Stoica, "Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles," 2021.

[240] P. Garrison, "Why is infiniband support important?" https://www.nimbix.net/why-is-infiniband-support-important/.

[241] G. Faanes, A. Bataineh, D. Roweth, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, J. Reinhard et al., "Cray Cascade: A scalable HPC system based on a Dragonfly network," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012, pp. 103:1–103:9.

[242] H. Kung, T. Blackwell, and A. Chapman, "Credit-based flow control for ATM networks: credit update protocol, adaptive credit allocation and statistical multiplexing," in *ACM SIGCOMM Computer Communication Review*, vol. 24, no. 4. ACM, 1994, pp. 101–114.

[243] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3098822.3098840 p. 239–252.

[244] Top500, "Top 500 Hpc systems," https://www.top500.org/.

[245] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young et al., "Mesh-tensorflow: Deep learning for supercomputers," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 435–10 444.

[246] "NvidiaDGX-1: The fastest deep learning system," https://devblogs.nvidia.com/dgx-1-fastest-deep-learning-system.

[247] "Cray in Azure," https://azure.microsoft.com/en-us/solutions/high-performance-computing/cray/.

[248] "Introducing the new HB and HC Azure VM sizes for HPC," https://azure.microsoft.com/en-us/blog/introducing-the-new-hb-and-hc-azure-vm-sizes-for-hpc/.

[249] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L. V. Kale, "Identifying the culprits behind network congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 113–122.

[250] A. BHATELE and L. V. KALE, "Quantifying network contention on large parallel machines," *Parallel Processing Letters*, vol. 19, no. 04, pp. 553–572, 2009. [Online]. Available: https://doi.org/10.1142/S0129626409000419

[251] M. Mubarak, P. Carns, J. Jenkins, J. K. Li, N. Jain, S. Snyder, R. Ross, C. D. Carothers, A. Bhatele, and K.-L. Ma, "Quantifying I/O and communication traffic interference on Dragonfly networks equipped with burst buffers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 204–215.

[252] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC applications under congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1013–1022.

[253] D. Skinner and W. Kramer, "Understanding the causes of performance variability in HPC workloads," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, 2005, pp. 137–149.

[254] "Monet," https://github.com/CSLDepend/monet.

[255] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini System Interconnect," in *2010 18th IEEE Symposium on High Performance Interconnects*, 2010, pp. 83–87.

[256] "CLOUD TPU: Train and run machine learning models faster than ever before." https://cloud.google.com/tpu/.

[257] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6D Mesh/Torus Interconnect for exascale computers," *Computer*, vol. 42, no. 11, p. 36–40, nov 2009. [Online]. Available: https://doi.org/10.1109/MC.2009.370

[258] Y. Ajima, T. Inoue, S. Hiramoto, S. Uno, S. Sumimoto, K. Miura, N. Shida, T. Kawashima, T. Okamoto, O. Moriyama, Y. Ikeda, T. Tabata, T. Yoshikawa, K. Seki, and T. Shimizu, "Tofu Interconnect 2: System-on-chip integration of high-performance interconnect," in *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Cham: Springer International Publishing, 2014, pp. 498–507.

[259] H. Miyazaki, Y. Kusano, N. Shinjou, F. Shoji, M. Yokokawa, and T. Watanabe, "Overview of the K computer system," *Fujitsu Sci. Tech. J*, vol. 48, no. 3, pp. 302–309, 2012.

[260] "Post-K supercomputer overview," http://www.fujitsu.com/global/Images/post-k-supercomputer-overview.pdf.

[261] C. Inc., "Managing system software for the Cray Linux Environment," Cray Doc S-2393-5202axx, 2014.

[262] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "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.

[263] S. Jha, A. Patke, M. Showerman, J. Enos, G. Bauer, Z. Kalbarczyk, R. Iyer, and W. Kramer, "Monet - Blue Waters Network Dataset," https://bluewaters.ncsa.illinois.edu/monet-bw-net-data/, University of Illinois at Urbana-Champaign, 2019.

[264] Theoretical and U. o. I. a. U.-C. Computational Biophysic Group, "NAMDv. 2.9 source code repository," http://www.ks.uiuc.edu/{D}evelopment/{D}ownload/download.cgi?PackageName=NAMD, 2014.

[265] M. D. Jones, J. P. White, M. Innus, R. L. DeLeon, N. Simakov, J. T. Palmer, S. M. Gallo, T. R. Furlani, M. Showerman, R. Brunner et al., "Workload analysis of Blue Waters," *arXiv preprint arXiv:1703.00924*, 2017.

[266] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q interconnection network and message unit," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–10.

[267] Z. Pang, M. Xie, J. Zhang, Y. Zheng, G. Wang, D. Dong, and G. Suo, "The TH Express high performance interconnect networks," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 357–366, 2014.

[268] W. J. Dally and C. L. Seitz, "Torus routing chip," jun˜12 1990, uS Patent 4,933,933.

[269] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable Dragonfly topology," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 77–88.

[270] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE transactions on Computers*, vol. 100, no. 10, pp. 892–901, 1985.

[271] J. Enos et al., "Topology-aware job scheduling strategies for torus networks," in *Proc. Cray User Group*, 2014.

[272] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. K. Panda, "Design of a scalable InfiniBand topology service to enable network-topology-aware placement of processes," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012, pp. 70:1–70:12.

[273] K. Pedretti, C. Vaughan, R. Barrett, K. Devine, and S. Hemmert, "Using the Cray Gemini Performance Counters," in *Proc. Cray User's Group*, 2013.

[274] C. Inc., "Managing network congestion in Cray XE Systems," Cray Doc S-0034-3101a Cray Private, 2010.

[275] J. Brandt, K. Devine, A. Gentile, and K. Pedretti, "Demonstrating improved application performance using dynamic monitoring and task mapping," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014, pp. 408–415.

[276] SchedMD, "Slurm scontrol," https://slurm.schedmd.com/scontrol.html.

[277] "Topology aware scheduling," https://bluewaters.ncsa.illinois.edu/topology-aware-scheduling.

[278] C. Inc., "Network resiliency for Cray XE and Cray XK Systems," Cray Doc S-0032-B Cray Private, 2013.

[279] C. D. Martino, S. Jha, W. Kramer, Z. Kalbarczyk, and R. K. Iyer, "Logdiver: a tool for measuring resilience of extreme-scale systems and applications," in *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*. ACM, 2015, pp. 11–18.

[280] Y. Sun, G. Zheng, L. V. Kale, T. R. Jones, and R. Olson, "A ugni-based asynchronous message-driven runtime system for cray supercomputers with gemini interconnect," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 751–762.

[281] T.-T. Lin and D. Siewiorek, "Error log analysis: statistical modeling and heuristic trend analysis," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 419–432, 1990.

[282] P. Garcia, F. Quiles, J. Flich, J. Duato, I. Johnson, and F. Naven, "Efficient, scalable congestion management for interconnection networks," *IEEE Micro*, vol. 26, no. 5, pp. 52–66, 2006.

[283] T. Rabbani, F. van den Heuvel, and G. Vosselman, "Segmentation of point clouds using smoothness constraint," *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 36, 01 2006.

[284] "Point Cloud Library," http://pointclouds.org.

[285] "Color-based region growing segmentation," http://pointclouds.org/documentation/tutorials/region_growing_rgb_segmentation.php.

[286] S. Jha, J. Brandt, A. Gentile, Z. Kalbarczyk, and R. Iyer, "Characterizing supercomputer traffic networks through link-level analysis," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 562–570.

[287] National Center for Supercomputing Applications, "SPP-2017 benchmark codes and inputs," https://bluewaters.ncsa.illinois.edu/spp-benchmarks.

[288] "Charm++ MiniApps," http://charmplusplus.org/benchmarks/\#amr.

[289] V. der Auwera et al., "From FastQ data to high-confidence variant calls: The genome analysis toolkit best practices pipeline," *Current protocols in bioinformatics*, pp. 11–10, 2013.

[290] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu, "A large scale study of data center network reliability," in *Proceedings of the Internet Measurement Conference 2018*. ACM, 2018, pp. 393–407.

[291] J. J. Galvez, N. Jain, and L. V. Kale, "Automatic topology mapping of diverse large-scale parallel applications," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079079.3079104 pp. 17:1–17:10.

[292] D. L. Donoho, "Breakdown properties of multivariate location estimators," Technical report, Harvard University, Boston. URL http://www-stat. stanford. edu/d̃onoho/Reports/Oldies/BPMLE. pdf, Tech. Rep., 1982.

[293] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[294] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing performance variations in HPC applications using machine learning," in *High Performance Computing*, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds. Cham: Springer International Publishing, 2017, pp. 355–373.

[295] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch out for the bully! job interference study on dragonfly network," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 750–760.

[296] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A multiplatform study of I/O behavior on petascale supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 33–44.

[297] 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: USENIX Association, Feb. 2019. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/yang pp. 379–394.

[298] A. Bhatele, N. Jain, Y. Livnat, V. Pascucci, and P.-T. Bremer, "Analyzing network health and congestion in Dragonfly-based supercomputers," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 93–102.

[299] N. Jain, A. Bhatele, X. Ni, N. J. Wright, and L. V. Kale, "Maximizing throughput on a Dragonfly network," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 336–347.

[300] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2934872.2934906 p. 101–114.

[301] V. Nathan, S. Narayana, A. Sivaraman, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Demonstration of the marple system for network performance monitoring," in *Proceedings of the SIGCOMM Posters and Demos*. ACM, 2017, pp. 57–59.

[302] M. Deveci, S. Rajamanickam, V. J. Leung, K. Pedretti, S. L. Olivier, D. P. Bunde, U. V. Çatalyürek, and K. Devine, "Exploiting geometric partitioning in task mapping for parallel computers," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 27–36.

[303] R. Grant, K. Pedretti, and A. Gentile, "Overtime: A tool for analyzing performance variation due to network interference," in *Proc. of the 3rd Workshop on Exascale MPI*, 2015.

[304] J. Brandt, E. Froese, A. Gentile, L. Kaplan, B. Allan, and E. Walsh, "Network performance counter monitoring and analysis on the Cray XC platform," in *Proc. Cray User's Group*, 2016.

[305] J. Brandt, K. Devine, and A. Gentile, "Infrastructure for in situ system monitoring and application data analysis," in *Proc. Wrk. on In Situ Infrastructures for Enabling Extreme-scale Analysis and Viz.*, 2015.

[306] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1879141.1879175 p. 267–280.

[307] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao et al., "The Sunway TaihuLight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, no. 7, p. 072001, 2016.

[308] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale RDMA deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 523–536, 2015.

[309] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats et al., "TIMELY: RTT-based congestion control for the datacenter," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 537–550.

[310] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/tammana pp. 233–248.

[311] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with switchpointer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/tammana pp. 453–456.

[312] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, "Compiling path queries," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, mar 2016. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/narayana pp. 207–222.

[313] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao et al., "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 479–491.

[314] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/handigol pp. 71–85.

[315] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 3–14.

[316] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[317] S. Floyd, "Congestion control principles," BCP 41, RFC 2914, September, Tech. Rep., 2000.

[318] S. Chunduri, T. Groves, P. Mendygral, B. Austin, J. Balma, K. Kandalla, K. Kumaran, G. Lockwood, S. Parker, S. Warren, N. Wichmann, and N. Wright, "GPCNeT: Designing a benchmark suite for inducing and measuring contention in HPC networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356215

[319] T. Hauer, P. Hoffmann, J. Lunney, D. Ardelean, and A. Diwan, "Meaningful availability," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, feb 2020. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/hauer pp. 545–557.

[320] M. Snir, W. D. Gropp, and P. Kogge, "Exascale research: preparing for the post-Moore era," 2011.

[321] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee, "Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence," 2 2019.

[322] G. K. Lockwood, N. J. Wright, S. Snyder, P. Carns, G. Brown, and K. Harms, "Tokio on clusterstor: Connecting standard tools to enable holistic i/o performance analysis," 2018.

[323] 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 Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2785956.2787496 p. 139–152.

[324] "splunk," https://www.splunk.com.

[325] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Dynamic fault-tree models for fault-tolerant computer systems," *IEEE Transactions on reliability*, vol. 41, no. 3, pp. 363–377, 1992.

[326] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar, "BASIL: Automated IO load balancing across storage devices," in *8th USENIX Conference on File and Storage Technologies (FAST 10)*. San Jose, CA: USENIX Association, feb 2010. [Online]. Available: https://www.usenix.org/conference/fast-10/basil-automated-io-load-balancing-across-storage-devices

[327] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[328] S. Jha, M. Showerman, A. Saxton, J. Enos, G. Bauer, B. Bode, J. Brandt, A. Gentile, Z. Kalbarczyk, R. K. Iyer, and W. Kramer, "Holistic measurement-driven system assessment," in *Proceedings of the ACM International Conference on Supercomputing*, Sep. 2019.

[329] "Operational data analytics," https://sc19.supercomputing.org/proceedings/bof/bof_pages/bof137.html, 2019.

[330] 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, FEB 2019.

[331] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. New York, NY, USA: Association for Computing Machinery, 2007. [Online]. Available: https://doi.org/10.1145/1282380.1282383 p. 13–24.

[332] W. M., H. S., S. K., G. T., and T. M., "CanarIO: Sounding the alarm on IO-related performance degradation," in *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium, IPDPS,*, 2020.

[333] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically finding the cause of packet drops," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, apr 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/arzani pp. 419–435.

[334] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, , 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, APR 2018.

[335] 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*. IEEE, 2018, pp. 108–121.

[336] 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.

[337] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[338] 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*. IEEE, 2002, pp. 595–604.

[339] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, apr 2012. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/nagaraj pp. 353–366.

[340] K. Khlebnikov and K. Kolyshkin, "ioping: simple disk I/O latency measuring tool," https://github.com/koct9i/ioping, 2017.

[341] T. Sherman, "Cray sonexion™ data storage system."

[342] "Lustre filesystem," http://lustre.org/, accessed: 2019-02-06.

[343] S. Oral and F. Baetke, "Lustre community BOF: Lustre in HPC and emerging data markets: Roadmap, features and challenges," https://sc18.supercomputing.org/proceedings/bof/bof_pages/bof176.html, 2018.

[344] M. Holland and G. Gibson, "Parity declustering for continuous operation in redundant disk arrays," in *Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-V)*, Boston, MA, USA, OCT 1992.

[345] "Zipkin," https://zipkin.io.

[346] Y. Shkuro, "Evolving Distributed Tracing at Uber Engineering," https://eng.uber.com/distributed-tracing/.

[347] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Trans. Storage*, vol. 7, no. 3, oct 2011. [Online]. Available: https://doi.org/10.1145/2027066.2027068

[348] L. Ma, T. He, A. Swami, D. Towsley, and K. K. Leung, "Network capability in localizing node failures via end-to-end path measurements," *IEEE/ACM transactions on networking*, vol. 25, no. 1, pp. 434–450, 2016.

[349] 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*, ser. IMC '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2663716.2663723 p. 195–208.

[350] B. D. Semeraro, R. Sisneros, J. Fullop, and G. H. Bauer, "It takes a village: Monitoring the Blue Waters supercomputer," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014, pp. 392–399.

[351] M. Butler, "Blue Waters super system," https://www.globusworld.org/files/2010/02/SuperSystem-BW-.pdf.

[352] D. Koller, N. Friedman, and F. Bach, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[353] P. Diaconis, D. Ylvisaker et al., "Conjugate priors for exponential families," *The Annals of statistics*, vol. 7, no. 2, pp. 269–281, 1979.

[354] R. M. Neal, "Probabilistic inference using Markov chain Monte Carlo methods," *Technical Report CRG-TR-93-1, Dept. of Computer Science, University of Toronto*, 1993.

[355] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, "Probabilistic programming in Python using PymC3," *PeerJ Computer Science*, vol. 2, p. e55, apr 2016. [Online]. Available: https://doi.org/10.7717/peerj-cs.55

[356] N. Taerat, J. Brandt, A. Gentile, M. Wong, and C. Leangsuksun, "Baler: Deterministic, lossless log message clustering tool," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 285, 2011.

[357] F. Mahdisoltani, I. Stefanovici, and B. Schroeder, "Improving storage system reliability with proactive error prediction," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USA: USENIX Association, 2017, p. 391–402.

[358] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. IEEE, 2003, pp. 119–126.

[359] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, and Z. Li, "deTector: A topology-aware monitoring system for data center networks," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, San Jose, CA, July 2017.

[360] H. Herodotou, B. Ding, S. Balakrishnan, G. Outhred, and P. Fitter, "Scalable near real-time failure localization of data center networks," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2623330.2623365 p. 1689–1698.

[361] D. Ghita, H. Nguyen, M. Kurant, K. Argyraki, and P. Thiran, "Netscope: Practical network loss tomography," in *Proceedings of 2010 IEEE Conference on Computer Communications (INFOCOM'10)*, San Diego, CA, USA, MAR 2010.

[362] "Parallel file system I/O benchmark," https://github.com/LLNL/ior.

[363] Z. C. Lipton, "The mythos of model interpretability," *Queue*, vol. 16, no. 3, pp. 31–57, 2018.

[364] 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, November 2016.

[365] E. B. Boyer, M. C. Broomfield, and T. A. Perrotti, "GlusterFS one storage server to rule them all," 7 2012.

[366] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, California, USA, JAN 2002.

[367] 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*. IEEE, 2006, pp. 31–31.

[368] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3131365.3131375 p. 78–85.

[369] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, "Detecting failures in distributed systems with the Falcon Spy Network," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2043556.2043583 p. 279–294.

[370] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish, "Taming uncertainty in distributed systems with help from the network," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741976

[371] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish, "Improving availability in distributed systems with failure informers," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, Lombard, IL, USA, APR 2013.

[372] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and enhancing in situ system observability for failure detection," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/huang pp. 1–16.

[373] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The /spl phi/ accrual failure detector," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, 2004, pp. 66–78.

[374] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, p. 225–267, march 1996. [Online]. Available: https://doi.org/10.1145/226643.226647

[375] M. K. Aguilera, G. Le Lann, and S. Toueg, "On the impact of fast failure detectors on real-time fault-tolerant systems," in *Distributed Computing*, D. Malkhi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 354–369.

[376] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive realtime datacenter fault detection and localization," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Boston, MA, MAR 2017.

[377] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. J. Wright, "A year in the life of a parallel file system," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 931–943.

[378] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, "Av-fuzzer: Finding safety violations in autonomous driving systems," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 25–36.

[379] B. Jafary, S. Jha, L. Fiondella, and R. K. Iyer, "Data-driven application-oriented reliability model of a high-performance computing system," *IEEE Transactions on Reliability*, pp. 1–13, 2021.