arXiv:2409.07774v1 [cs.SE] 12 Sep 2024

ROCAS: Root Cause Analysis of Autonomous Driving Accidents via Cyber-Physical Co-mutation

Shiwei Feng
Purdue University
West Lafayette, USA
feng292@purdue.edu

Yapeng Ye
Purdue University
West Lafayette, USA
ye203@purdue.edu

Qingkai Shi*
The State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
qingkaishi@nju.edu.cn

Zhiyuan Cheng
Purdue University
West Lafayette, USA
cheng443@purdue.edu

Xiangzhe Xu
Purdue University
West Lafayette, USA
xu1415@purdue.edu

Siyuan Cheng
Purdue University
West Lafayette, USA
cheng535@purdue.edu

Hongjun Choi
DGIST
Daegu, South Korea
hongjun@dgist.ac.kr

Xiangyu Zhang
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

ABSTRACT

As Autonomous driving systems (ADS) have transformed our daily life, safety of ADS is of growing significance. While various testing approaches have emerged to enhance the ADS reliability, a crucial gap remains in understanding the accidents causes. Such post-accident analysis is paramount and beneficial for enhancing ADS safety and reliability. Existing cyber-physical system (CPS) root cause analysis techniques are mainly designed for drones and cannot handle the unique challenges introduced by more complex physical environments and deep learning models deployed in ADS. In this paper, we address the gap by offering a formal definition of ADS root cause analysis problem and introducing Rocas, a novel ADS root cause analysis framework featuring cyber-physical co-mutation. Our technique uniquely leverages both physical and cyber mutation that can precisely identify the accident-trigger entity and pinpoint the misconfiguration of the target ADS responsible for an accident. We further design a differential analysis to identify the responsible module to reduce search space for the misconfiguration. We study 12 categories of ADS accidents and demonstrate the effectiveness and efficiency of Rocas in narrowing down search space and pinpointing the misconfiguration. We also show detailed case studies on how the identified misconfiguration helps understand rationale behind accidents.

ACM Reference Format:
Shiwei Feng, Yapeng Ye, Qingkai Shi, Zhiyuan Cheng, Xiangzhe Xu, Siyuan Cheng, Hongjun Choi, and Xiangyu Zhang. 2024. ROCAS: Root Cause Analysis of Autonomous Driving Accidents via Cyber-Physical Co-mutation. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3691620.3695530

*Corresponding author.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695530

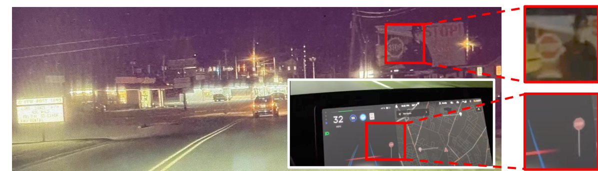


Figure 1: A real-world emergency braking case reported by [7], as the Tesla autopilot recognizes the stop sign from the road-side billboard. Billboards are common and *benign*, but such scenarios can be *accident-inducing*.

1 INTRODUCTION

Autonomous driving has achieved remarkable breakthroughs [34, 59] and becomes closer and closer to our daily life [41, 82]. From self-driving cars to delivery robots, autonomous driving techniques are revolutionizing the way we live and work. A typical modern autonomous driving system (ADS) employs perception modules to interpret and understand the surrounding environment, prediction and planning algorithms to interact with other vehicles, and controllers to maintain stability and propulsion. These modules can each be incorporated with Deep Learning algorithms, which enhance the ADS's intelligence and adaptability over time.

However, as with any complex system, ADS are always prone to errors, which may lead to runtime failures. Due to the inherent uncertainty in Deep Learning models, the complexity of the physical world, and the imprecision in control software [23], ADS accidents have been witnessed [9–12, 15], many of them having devastating consequences. However, the underlying reasons behind these accidents are not readily apparent. For example, as shown in Figure 1, the Tesla autopilot recognizes the vague stop sign pattern from a road-side billboard, leading to an emergency braking. While the billboard means no harm, such a scenario can be confusing to ADS and potentially induce accidents. Therefore, post-accident analysis that identifies accident causes is of increasing importance for ADS companies and developers to improve ADS safety and reliability.

Traditionally, post-accident analysis has been performed either in the physical domain, e.g., physical crime scene investigation [60], or in the cyber domain, focusing on disclosing trails and provenance of cyber attacks [48, 65, 86]. However, ADS is essentially a cyber-physical system (CPS) that requires co-analysis of both the

cyber and physical worlds. While there are a number of pioneering post-accident analysis techniques in CPS domain, they mainly focus on drone systems rather than ADS. For instance, MAYDAY [53] employs program analysis to diagnose accidents caused by controller bugs and mission command bugs using a pre-constructed dependency graph between controllers. RVPlayer [22] decouples aggregated environmental disturbances during logging and applies them to drones for faithful replay. Although these techniques are effective in their targeted scopes, they can hardly be applied to the ADS domain. Firstly, ADS introduces more complex modules such as perception, prediction, and planning modules. MAYDAY relying on the domain specific knowledge of controller programs cannot support other complex modules in ADS. Furthermore, deep learning models are widely used in ADS, which introduce a significant amount of inherent uncertainty for such traditional program analysis based techniques. Secondly, ADS operates in much more complicated and interactive physical environments, including a lot of external entities such as other vehicles, pedestrians, traffic lights. Thus, it is not straightforward for RVPlayer to decouple and reapply these *indirect* environment influences, and thus the accident replay on ADS is highly challenging.

As far as we know, there is no existing root cause analysis designed for ADS. In this paper, we define ADS root cause analysis as a post-accident analysis to identify the *triggering entity* (e.g., external physical objects) and the *misconfiguration* (e.g., configurable parameters used by ADS) that causes the accident. We propose Rocas, a novel ADS root cause analysis framework via cyber-physical comutation. Given an accident, our technique can precisely pinpoint the triggering entity and the misconfiguration of the target ADS. Specifically, Rocas first faithfully replay the accident execution inside the simulator, using the recorded locations of ADS and other entities during runtime. Then it performs *physical mutation* to identify the trigger entity, by finding the minimal environmental entity mutation that suppresses the accident, without changing the ADS's configuration. Finally, Rocas conducts *cyber mutation* to pinpoint the misconfiguration, by searching for the minimal ADS configuration mutation, without changing the ADS's trajectory before the accident. In practice, cyber mutation proves to be highly time-consuming, primarily due to two reasons. Firstly, the search space is extensive, as exemplified by Baidu Apollo [2], which encompasses over 1100 configurations. Deciding which subsets of configurations to mutate and determining the magnitude of the mutated values requires considerable effort. Secondly, this search process is not easily parallelizable. Despite the availability of a decent GPU with 8 GB graphical memory, it can only run one simulator and one ADS concurrently. As a result, we further propose a differential analysis algorithm on ADS execution records to reduce the search space for cyber mutation. Details can be found in Section 5. Our contributions are summarized as follows.

- We formally define the problem of ADS root cause analysis and propose Rocas, a novel ADS root cause analysis framework, which incorporates both physical and cyber mutation. These techniques accurately identify the *triggering entity* and pinpoint the *misconfiguration*, respectively.
- We introduce a differential analysis algorithm that effectively reduces the search space for the misconfiguration by comparing two execution records.
- We implement a prototype of Rocas and evaluate its effectiveness on 12 different types of ADS accidents, including a total of 144 accident cases. Through extensive experiments, we demonstrate that Rocas can precisely pinpoint the misconfiguration responsible for each accident with high efficiency. The code, video demos, and supplementary materials are available at [6].

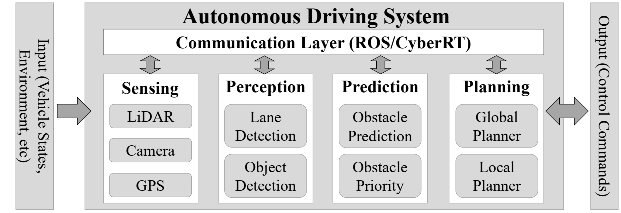


Figure 2: A general architecture of ADS with major modules.

2 BACKGROUND

We introduce the typical ADS architecture (Section 2.1) and ADS communication framework (Section 2.2).

2.1 ADS Architecture

ADS architecture contains multiple modules, while the four most closely linked to ADS decision-making are: sensing, perception, prediction, and planning, as shown in Figure 2.

These modules operate in a cascaded fashion, transforming from the initial hardware sensor input (e.g., video stream and LiDAR point cloud) to the final driving controls (e.g., steering, acceleration, and braking). Specifically, (1) **Sensing** utilizes a variety of sensors (cameras, LiDARs, RADARs, etc.) to gather raw environmental data. (2) **Perception** interprets this data to understand the environment using deep learning for tasks like traffic sign recognition and object tracking. (3) **Prediction** assesses current object statuses to predict future movements and prioritize actions, aiding in accident avoidance. (4) **Planning** comprises a global planner for route determination and a local planner for real-time trajectory adjustments based on environmental conditions. (5) **Control** executes the driving plan by controlling vehicle movements and adjusting to real-time conditions to ensure adherence to the trajectory.

Each module has hundreds of configurable parameters, such as the obstacle buffer or the reactive braking distance. While these configurations add flexibility to ADS, they simultaneously increase the difficulty of testing and debugging.

2.2 ADS Communication Framework

ADS relies on communication frameworks for data transmission among modules, which help decouple the functions of diverse modules and facilitate system development. ADS communication frameworks are typically based on Robot Operating System (ROS). ROS employs a publish-and-subscribe mechanism, whereby each *ROS node* can publish data under a specific topic, ensuring that all nodes subscribing to this topic obtain the published data. For example, Baidu Apollo [2], as one of the most popular ADS, utilized ROS as its communication framework in the first few versions and later developed CyberRT[3] based on ROS, with optimized efficiency.

3 MOTIVATION

This section presents an accident example to illustrate ADS root cause analysis challenges and showcase our technique.

3.1 Motivating Example

LAV. LAV [21] is an open-source ADS that won the champion of the 2021 CARLA AD Challenge [13]. Figure 3 depicts its high-level code logic. It is a representative design of a modularized ADS, following the diagram of *perception-prediction-planning-control*. The function `run_step()` at Line 4 is regularly executed inside the main loop at a frequency of 20Hz. At Lines 6-9, the ADS retrieves location information (e.g., `gps`), perception raw data (e.g., `lidar` and `tel_rgb`), and vehicle states (e.g., `speed`) from sensors. The location information `gps` is used to determine the road option `opt`, e.g., turning right or moving forward (Line 13). LiDAR is used to plan ADS's trajectory `ego_plan_traj` and predict other vehicles' trajectories `other_pred_traj` (Lines 15-16). Telephoto images `tel_rgb` are fed into `brake_model()` to obtain a braking probability `pred_brake` based on traffic lights and hazard conditions (Line 18). The controller `pid_control()` leverages the processed data to produce control commands (i.e., `steer`, `throt` and `brake`) at Lines 22-24. Additionally, post-processings are necessary to ensure adherence to the constraints, such as collision avoidance (function `plan_collide`), hazard stop (`BRAKE_THRESHOLD`), and speed limit (`MAX_SPEED`) at Lines 25-32.

Accident Example. Figure 4 illustrates the scenario wherein an emergent braking accident is unexpectedly triggered. In this instance, the ADS (shown as the blue car) is following the mission to navigate through the roundabout. However, during its execution, the AD vehicle unexpectedly comes to a stop within the roundabout (at the red point). This unforeseen emergency braking within the roundabout could lead to serious rear-end collisions.

Accident Explanation. In this example, a red truck ahead ADS triggers a misconfiguration `BRAKE_THRESHOLD` and leads to an accident. As shown at Line 27 in Figure 3, the variable `pred_brake` is compared with a fixed threshold `BRAKE_THRESHOLD`, whose value is set to 0.1 at Line 2, to determine if a brake is necessary. `pred_brake` means the probability of braking, computed by a DL model `brake_model` (Line 18) which takes as input the images from ADS's camera with telephoto lens . Figure 5a shows the captured images when the AD vehicle stops within the roundabout. In normal cases, without a red light, the `brake_model()` outputs a value (i.e., `pred_brake`) close to zero, which is smaller than the `BRAKE_THRESHOLD`. However, in this example, there is a red struck positioned at this particular angle, which shares partial feature with a red traffic light and makes the variable `pred_brake` fluctuate around 0.2. Note that this is still a relatively small braking probability. However, after comparison with the fixed threshold, the variable `brake` is set to 1 at Line 28 and passed to the controller at Line 34, resulting in an emergency braking.

3.2 Limitation of Existing Works

After an accident happens, post-accident analysis must be conducted to identify the underlying root causes, including both the external triggering entities (e.g., the red truck in Figure 4) and the internal misconfigurations (e.g., the variable `pred_bake`), and improve the reliability of the ADS.

Unlike traditional software root cause analysis, ADS is a cyber-physical system (CPS), which requires the joint analysis of both the cyber and physical domains. Existing CPS post-accident analysis frameworks mainly focus on drone systems. There are two main types of techniques, i.e., program analysis based and what-if reasoning based. These post-accident analysis for drone systems usually focus on the controller components and primarily address simple environmental factors such as weather conditions (e.g., wind gust). However, ADS has a much more complex architecture, equipped with sophisticated modules such as perception, prediction, and planning modules, and also with deep learning models. In addition, ADS operates in much more complicated physical environments, including external entities such as vehicles, pedestrians, and traffic lights. These factors make it difficult to directly apply existing methods for drone systems to ADS scenarios.

Program analysis based method. MAYDAY [53] utilizes program analysis to help diagnose accidents caused by controller bugs and mission command bugs. It leverages a pre-constructed dependency graph between controllers to find the state deviation. Thus, MAYDAY can only identify a potentially problematic basic block in control program, without pinpointing a specific line of code or a configuration. Also, it is based on the domain specific knowledge of controller programs and cannot support other complex modules in ADS, such as the perception, prediction, and planning. Furthermore, deep learning models are widely used in ADS, which introduce a significant amount of inherent uncertainty for traditional program analysis based techniques.

What-if reasoning based method. Due to the limitation of program analysis based techniques, some works investigate the accident root causes by storing the environmental disturbances and replaying the execution. RVPlayer [22] decouples the aggregated environmental disturbances during logging and reapplies this disturbance on drones to enable faithful replay. However, ADS operates in much more complicated and interactive physical environments, including a lot of external entities such as other vehicles, pedestrians, traffic lights. Different from drone systems that are primarily *reactive* to environmental disturbances, ADS is required to be *proactive* to handle these intricate situations. Thus, it is not straightforward to decouple and reapply these indirect environment influences, and the accident replay on ADS is highly challenging.

3.3 Our Approach

In this paper, we propose an ADS root cause analysis featuring *cyber-physical co-mutation* to identify the accident causes. To apply the mutation, we first need to reconstruct the accident in a simulator (we call it the accident execution). We instrument the ADS to record its own locations from GPS and the bounding box (including relative locations with respect to ADS) of other vehicles. Given the map file, all the locations can be transformed into simulator locations, enabling the simulator to replay the accident execution.

After replaying the accident execution in the simulator, we apply two steps of mutation to identify the triggering entities and misconfigurations, separately. First, we keep all ADS configurations (i.e., the cyber space) unchanged, only mutate the physical conditions,

```
1 const MAX_SPEED = 35 // km/h
2 const BRAKE_THRESHOLD = 0.1
3 MISSION = read_config_file()
4 def run_step(input_data) {
5     // Get sensor data and vehicle states
6     gps = input_data.get('GPS')
7     lidar = input_data.get('LIDAR')
8     tel_rgb = input_data.get('TEL_RGB')
9     speed = input_data.get('ego_speed') // m/s
10    ...
11    // Get road option from global planner,
12    // e.g., RIGHT/LEFT/FORWARD/STOP.
13    opt = waypointer(MISSION, gps)
14    // Motion prediction & local planning
15    ego_plan_traj, other_pred_traj = \
16                    infer_model(lidar, opt)
17    // Brake prediction from telephoto lens images
18    pred_brake = brake_model(tel_rgb)
19    ...
20    // Control command
21    steer, throt, brake = 0, 0, 0
22    if not has_none_val(ego_plan_traj):
23        steer, throt, brake = \
24            pid_control(ego_plan_traj, speed, opt)
25    collide_flag = plan_collide( \
26        ego_plan_traj, other_pred_traj)
27    if pred_brake > BRAKE_THRESHOLD:
28        throt, brake = 0, 1
29    elif collide_flag:
30        throt, brake = 0, 1
31    if speed * 3.6 > MAX_SPEED:
32        throt = 0
33    ...
34    return steer, throt, brake
35 }
```

Figure 3: High-level Code Logic of LAV [21]

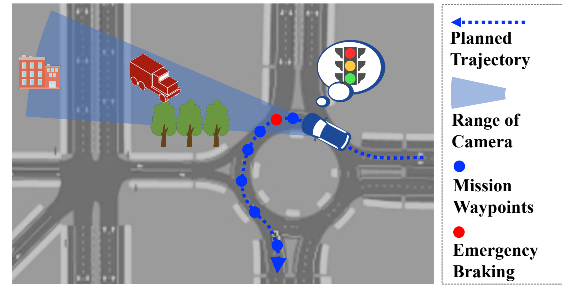


Figure 4: An accident example for LAV. The vehicle unexpectedly stops within the roundabout, triggered by the red truck ahead.

(a) Emergency Braking.

(b) No Emergency Braking.

Figure 5: Subfig (a) shows the image captured by ADS's cameras when it stops within the roundabout. Subfig (b) removes the red truck in the middle region and prevents the accident.

replay the execution in the simulator, and observe if the accident still occurs. We call this process as *physical mutation*, aiming to find the minimal mutation (in physical space) to suppress the accident. The example in Section 3.1 demonstrates an emergency braking accident. It is possible that a specific physical condition may fool deep learning models used by ADS and cause them to mistake a red light or an obstacle on the path, resulting in an emergency braking decision. Figure 5a displays the images captured by the ADS's camera at the emergency braking point within the roundabout. If we disable the red truck appearing in the middle region of the cameras (as shown in Figure 5b), while keeping all configurations and other physical conditions unchanged, the subject vehicle will not stop within the roundabout during replay. Thus, we know that the red truck is the triggering entity that leads to the accident.

In the second mutation, we search for the misconfiguration. Unlike physical mutation, in this step, we only mutate the configurations inside ADS, while keeping the physical space. The mutation in this process is referred to as *cyber mutation*. This mutation aims to find the minimal mutation (in cyber space) to suppress the accident and outputs the misconfigurations that lead to the accident. For example, there are two configurations shown in Figure 3, i.e., `MAX_SPEED` and `BRAKE_THRESHOLD` (Line 1-2). When we increase the value of `BRAKE_THRESHOLD` to 0.5, the accident is suppressed during the replay, and the subject vehicle can successfully pass through the roundabout. However, if we mutate another configuration, such as `MAX_SPEED` (e.g., set its value to 50), the accident still occurs. This reveals that `BRAKE_THRESHOLD` is highly likely to be the misconfiguration and helps us identify Line 27, which is responsible for the accident.

However, a challenge is that cyber mutation can be very time-consuming due to the large search space of configurations. To reduce search space, we identify the initial deviating module that is likely to cause the accident before searching for misconfigurations in cyber mutation. Specifically, we record *the channel messages communicated among modules* to represent an execution record. Then, we conduct differential analysis on the reference execution records (obtained from physical mutation) and the accident execution records, to facilitate the initial deviating module.

To summarize, we propose RoCAS, an ADS root cause analysis framework for post-accident analysis. The overall system design of RoCAS is shown in Figure 6.

In Phase-I, given an accident, RoCAS replays the accident execution in simulation. Then in Phase-II, RoCAS conducts physical mutation on the accident execution, producing the triggering entity and an accident-free reference execution. In Phase-III, by doing differential analysis on the two execution records from accident execution and reference execution, RoCAS identifies the initial deviating module in order to reduce search space for Phase-IV. Finally in Phase-IV, RoCAS runs cyber mutation only within the identified module and outputs the misconfiguration. More details are elaborated in Section 5.

4 DEFINITION OF ADS ROOT CAUSE ANALYSIS

We formally define ADS execution and accident root cause analysis.

4.1 ADS Execution

An execution takes as input a configurable ADS and physical environment (e.g., map, weather, and other vehicles). The execution records how the ADS reacts to the environment and reports whether an accident happens during the execution.

Configurable ADS. Formally, we denote the type of the cyber configuration as $C \coloneqq C_1 \times C_2 \times \cdots \times C_r$, where $C_i \coloneqq \mathbb{R}^{\mu_i}$ denotes the type of the configuration for module i.

Physical environment. Physical environment consists of two parts: the actor entities (e.g., moving vehicles) and the map entities (e.g., traffic cones, traffic lights, and buildings). Formally, we use $\mathcal{A} \coloneqq \mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_j$ and $\mathcal{M} \coloneqq \mathcal{M}_1 \times \mathcal{M}_2 \times \cdots \times \mathcal{M}_k$, to denote the type of the states for actor entities and map entities, respectively. Specifically, supposing that each entity has σ properties (e.g., location, orientation, velocity), $\mathcal{A}_i \coloneqq \mathbb{R}^\sigma$ and $\mathcal{M}_i \coloneqq \mathbb{R}^\sigma$ denotes the type of the states the ith actor entity and that for the ith map entity, respectively.

ADS execution. One complexity of ADS execution is that properties of an actor entity may change over time. We thus utilize $A_i@t$, where $A_i \in \mathcal{A}_i$, to represent the state of the i-th actor entity at timestamp t. Moreover, we adopt $A_i@[t_1, t_2]$ to denote a sequence of states of the i-th actor entity during the time span $[t_1, t_2]$. For simplicity, we use $A@t$, where $A_i \in \mathcal{A}$, to denote the state of all actor entities at the timestamp t and $A@[t_1, t_2]$ the state sequences of all actor entities during the time span $[t_1, t_2]$.

Formally, we use $\mathcal{E}$ to denote the type of an execution: $\mathcal{E} : C \times \mathcal{M} \times \mathcal{A} \to \text{List}[\mathcal{A}] \times \mathbb{B}$. Intuitively, an execution takes as input the configuration of the ADS, the states of the map entities and actor entities, and outputs the states of every actor entity at all timestamps. Moreover, it also outputs a boolean value indicating whether an accident happens (i.e., collision or emergency braking).

To facilitate discussion, we use C to denote a concrete instance of the cyber configuration; Similarly, A for actor entities, M for map entities and E for an execution. Note that we assume the first element in A always denotes the AD vehicle, namely A_0.

4.2 ADS Accident Root Cause Analysis

Root cause analysis for ADS accident aims to identify the triggering entities and the misconfigurations. We formally define an accident execution as $E\Big(C, M, A@t_0\Big) \to \langle A@[t_0, t_0 + T], True\rangle$. We further assume the accident happens after a time period of d after starting.

Triggering entities identification. We define the triggering entities as a set of actor and map entities such that minimal changes to their states (noted as $\Delta M \in \mathcal{M}$ and $\Delta A \in \mathcal{A}$) would:
(1) *suppress the accident.* Formally,

$$E\Big(C, M \oplus \Delta M, A@t_0 \oplus \Delta A\Big) \to \langle A@[t_1, t_1 + T], False\rangle,$$

where $\oplus$ denotes applying changes to the states values.
(2) and *introduce limited changes to the actor entities before entering the accident scene.* Formally,

$$\Sigma_{t\in[0,d-\delta]}\Big(\big\|A@(t_0 + t) - A@(t_1 + t)\big\|\Big) < \epsilon,$$

where δ denotes a short time period before the accident occurs, and $\|\cdot\|$ denotes L^2 norm.

Table 1: Physical Mutation Space.

Domain	Category	Configuration	Data Type
Actor Entities	Vehicles	Model	{Sedan, Truck, ...}
		Color	{Red, Blue, Black, ...}
		Location	[x y z]
		Rotation	[yaw pitch roll]
		Speed	S (m/s)
	Cyclists/ Pedestrians	Location	[x y z]
		Rotation	[yaw pitch roll]
		Speed	S (m/s)
Map Entites	Traffic Cones/ Boxes	Location	[x y z]
		Rotation	[yaw pitch roll]
	Buildings	Enable	{True, False}
	Vegetations	Enable	{True, False}
	Traffic Lights	Policy	{Red, Yellow, Green}
		Enable	{True, False}
	Weather	Cloudiness	[0, 100]
		Precipitation	[0, 100]
		Sun Azimuth Angle	[0, 360] (deg)
		Sun Altitute Angle	[-90, 90] (deg)
		Fog Density	[0, 100]

Misconfiguration identification. After finding triggering entities, we further search for misconfigurations. Formally, we search for minimal changes of configurations, noted as $\Delta C \in C$, such that (1) *the accident is suppressed.* Formally,

$$E\Big(C \oplus \Delta C, M, A@t_0\Big) \Rightarrow \langle A@[t_2, t_2 + T], False\rangle,$$

where $\oplus$ denotes applying changes to the original configurations.
(2) *the AD vehicle does not change its behavior before entering the accident scene.* Formally,

$$\Sigma_{t\in[0,d-\delta]}\Big(\big\|A_0@(t_0 + t) - A_0@(t_2 + t)\big\|\Big) < \epsilon,$$

where δ denotes a short time period before the accident occurs and $\|\cdot\|$ denotes the L^2 norm. This requirement ensures the AD vehicle encounters a similar scenario before and after the configuration changes. Otherwise, one can always freeze the AD vehicle to suppress the accident, which is meaningless for the investigation.

5 SYSTEM DESIGN

In this section, we discuss the design details of Rocas. First, Rocas replays the accident execution by applying coordinate transformation on execution records. Then it conducts physical mutation (Section 5.1) to find the accident-triggering entities. Moreover, the physical mutation further yields a reference execution without accident. To search for the misconfigurations, Rocas first pinpoints the initial deviating module (Section 5.2) to reduce search space via differential analysis between the accident execution record and the reference execution record. Finally Rocas performs cyber mutation to identify the misconfigurations (Section 5.3).

5.1 Physical Mutation

Physical mutation serves two purposes: (1) it helps produce the accident root cause results by finding the accident-triggering entities; (2) it outputs a reference execution that is similar to the original execution but accident-free. An accident-free reference execution indicates that the accident could have been avoided in a similar scenario. The differential analysis in the later stage will compare

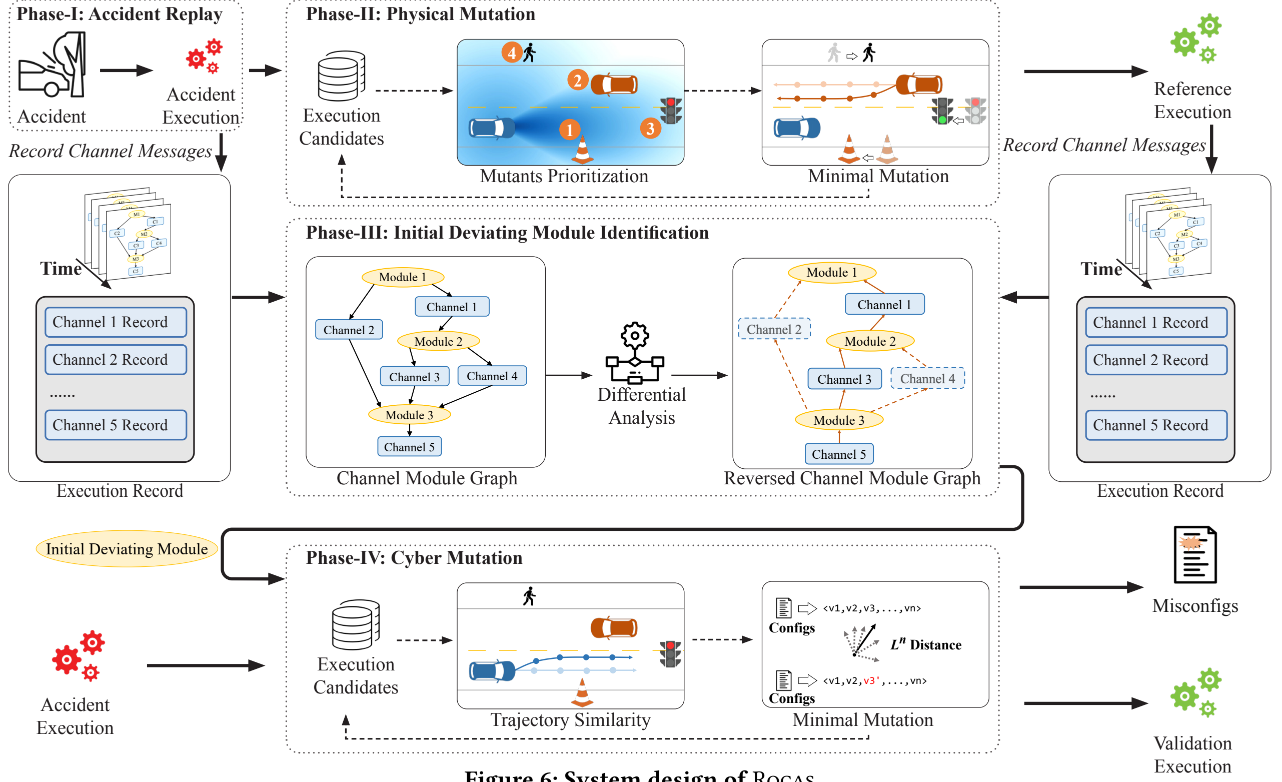


Figure 6: System design of Rocas

these two execution records to facilitate localizing the module that has accident-inducing behaviors.

To search for an accident-free execution, the physical mutation stage mutates the surrounding entities (i.e., A, M defined in Section 4.1) without modifying the configuration of the AD vehicle. The entity space, which is also Rocas's mutation space, is shown in Table 1.

We formulate the physical mutation stage as a Multi-object Optimization Problem (MOP) [27]. The MOP algorithm takes as input a set of objective functions, a set of constraints, and a set of variables from the solution domain. It employs a genetic algorithm and outputs an optimal set of variables with the minimal objective values and meanwhile satisfy the constraints. In this stage, we leverage the MOP algorithm to search for a set of properties for the surrounding entities with minimal changes to the original execution and eliminate the accident.

Our fitness function requires that (a) the replay execution yields false. (i.e., the accident is suppressed); (b) the mutated entity is highly related to the triggering condition; (c) the replay scenario should be similar to the original one; (d) the set of mutated physical conditions is small; and (e) their value changes are small.

Following the notations discussed in Section 4.1, $E(\cdot)$ returns whether the accident happens. Given an initial physical condition $p_0 \in C \times \mathcal{A} \times \mathcal{M}$ such that $E(p_0) = 1$, finding the triggering entity (i.e., minimum mutation) can be formulated as:

$$\begin{aligned} &\textbf{minimize } \mathcal{F}(p) = \{f_1(p), f_2(p), f_3(p)\}, \\ &\textbf{subject to } \mathcal{G}(p) = \{E(p) = 0\}. \end{aligned} \tag{1}$$

$\mathcal{F}$ represents the function we aim to minimize, while $\mathcal{G}$ denotes the constraint that must be satisfied. Specifically, $f_1(\cdot)$ defined in Eq. 2 quantifies the suspiciousness of the mutated entity and satisfies the requirement (b); $dist_i$ and θ_i refer to the distance and angle from mutated entity i to the ADS; K is a parameter that balances between distance and angle. Intuitively, the entity that is close to and in front of the ADS are more likely to be accident-inducing than those far away or behind the ADS. As visualized in the "Mutants Prioritization" figure in Figure 6 "Phase-I" block, darker blue denotes higher relevance, while white denotes not relevant.

$$f_1(p) = -\sum_{i \in p - p_0} \Big(K/dist_i + cos(\theta_i)\Big) \tag{2}$$

The $f_2(\cdot)$ satisfies the requirement (c) that the replay scenario should be similar to the original one. We quantify the scenario similarity using the trajectory similarity (computed by MSE, mean squared error) of all actor entities. $Traj_p(j)$ denotes the trajectory of actor entity j under the physical condition p. This objective function is necessary, otherwise we can just set all actor entities' speed to zero and the accident can be suppressed.

$$f_2(p) = -\sum_{j \in A} Sim\Big(Traj_{p_0}(j), Traj_p(j)\Big) \tag{3}$$

Function f_3 defines the distance of an offspring from the initial physical configuration p_0, satisfying the requirements (d) and (e). The first term measures that the set of mutated physical conditions, which should be small. The second and the third terms quantify that the value changes of mutated physical conditions should be small. We use L_1-Norm to compute the edit distance of enumerate

and boolean type configurations and use L_2-Norm for other types.

$$f_3(p) = \#(p - p_0) + \sum_{k \in Enum} L_1(k_0, k) + \sum_{k \notin Enum} L_2(k_0, k) \quad (4)$$

5.2 Initial Deviating Module Identification

After finding an accident-free reference execution, Rocas differentiates the reference execution and the original execution to localize the module that potentially has misconfigurations. There are two major challenges in differentiating two executions records. First, due to the noise in the physical world, even two executions with the same setup may have differences. It is thus important to distinguish the differences that lead to different behaviors of the ADS and the differences that are caused by noise. Moreover, since the reference execution and the original execution have slightly different setup, Rocas may observe differences in multiple modules of the system. For example, if a position of an actor car is slightly changed, both vision module and planning module may output different results. It is thus essential to locate the accident-inducing differences.

We propose to represent an ADS as a channel module graph (CMG), and use the messages in channels to represent an execution. Rocas differentiates the messages to locate the problematic module and traverse the CMG to identify the root cause.

5.2.1 Graph representation for ADS. Modules in an ADS use channels to communicate with each other. A channel represents a segment of shared memory within the ADS. A module reads messages from other modules via channels and writes its computational outcomes to output channels. Different modules run in parallel to achieve better real-time performance. Traditional program representations, like control flow graph or data dependency graph, cannot handle the *temporal* (e.g., variable values are updated tens or hundreds of times at each second) and *heterogeneous* (e.g., involve multiple modules and deep learning models) features of ADS. We thus propose channel module graph (CMG) defined in Definition 5.1 to represent ADS programs. We use an example from Apollo [2] to illustrate how Rocas leverages CMG to represent ADS structure.

Definition 5.1 (Channel Module Graph). Given an ADS, we use a directed bipartite graph, *Channel Module Graph (CMG)*, to represent ADS topology structure, denoted as $G = \langle V_C, V_M, E \rangle$.
(1) Two disjoint vertices set V_C and V_M represent *channels* and *modules* of ADS, respectively.
(2) $e = \langle v_M, v_C \rangle \in E, v_M \in V_M, v_C \in V_C \iff v_M$ *writes to* v_C.
(3) $e = \langle v_C, v_M \rangle \in E, v_M \in V_M, v_C \in V_C \iff v_M$ *reads from* v_C.

A concrete example is shown in Figure 7a. Note that the CMGs of the original execution and the reference execution are the same since the reference execution only mutates the surrounding environments without altering ADS code.

5.2.2 Message as execution records. We use messages that communicated in channels as the records of ADS execution, considering its two advantages: (1) the messages naturally capture the temporal feature of ADS since a typical message contains the timestamp. (2) the message is agnostic of the possibly heterogeneous implementation of multiple modules.

Figure 7b shows (simplified) message definition for the *planning* channel. Each planning channel message consists of (1) a `lane_id` field, denoting the lane that AD vehicle should be on; (2) a `traj_point` field, containing a sequence of trajectory points that represents the expected position of the AD vehicle. Due to the nested structure of messages, one piece of message can be easily transformed into its corresponding tree representation (Figure 7c).

5.2.3 Differential analysis on execution records. Recall that we now have two records, one obtained from the replay accident execution, the other from the accident-free reference execution. We conduct differential analysis on these two execution records to locate the responsible module for the accident occurrence.

We introduce the metric *Message Difference Ratio* (MDR) to quantify the difference between two messages. Given two execution records, Rocas computes MDR for each channel at each timestamp.

Definition 5.2 (Message Difference Ratio). Given two messages m_1, m_2 (with tree representations tr_1, tr_2), their *message difference ratio* (MDR) are computed as follows:

$$\mathtt{MDR}(tr_1, tr_2) = \begin{cases} 1 \textbf{ if } tr_1 \neq tr_2 \textbf{ else } 0, & |tr_1.chd| = |tr_2.chd| = 0 \\ 1, & |tr_1.chd| \neq |tr_2.chd| \\ \dfrac{\sum_{i=0}^{|tr_1.chd|} \mathtt{MDR}(tr_1.chd[i], tr_2.chd[i])}{|tr_1.chd|}, & Otherwise \end{cases}$$

Intuitively, MDR takes as input two trees, and outputs a difference score from 0 (same) to 1 (different). The first case means both input trees are leafs. The two leafs are then directly compared. The second case indicates that two input trees have different numbers of children. We simply consider them as different. For two non-leaf tree nodes with the same number of children, we recursively compare each child, and use the average MDR scores of all children as the MDR score of the two input trees.

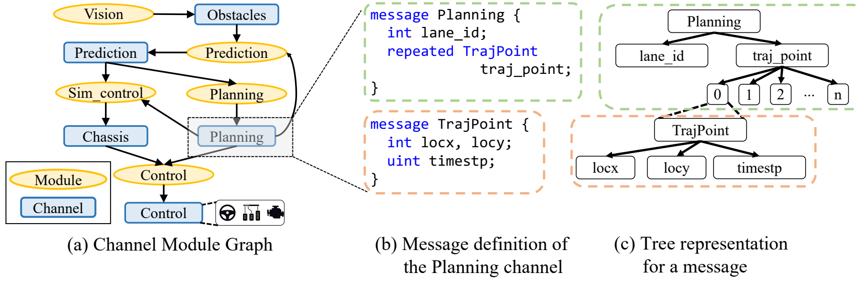
As the output of differential analysis, for each channel, we obtain a series of MDR at each timestamp. These MDR series are further leveraged to pinpoint the initial deviating module.

5.2.4 Pinpoint initial deviating module. Our insight is that: (1) *the timestamp that `control` channel has a sudden change of* MDR *is the occurrence time of the accident.* (2) *the channels that has sudden changes of* MDR *before the accident occurrence time are likely to be responsible for the accident.*

Rocas leverages the `PELT` [49] change point detection algorithm to detect when MDR changes significantly for each channel. The `PELT` algorithm detects significant change points from a sequence of values. The rationale is that the first few seconds of compared executions are expected to be similar, thus MDR in the first few seconds can be considered as the baseline noise of a channel. A significant change on MDR value indicates that the difference among channel messages significantly goes beyond noise.

Detailed pinpoint algorithm is shown in Alg. 1. It takes 2 inputs, the CMG G and the MDR series of all channels *channel2mdr* obtained from differential analysis in Section 5.2.3. Output is the identified initial deviating module that are responsible for the accident.

Specifically, given *channel2mdr*, Alg. 1 first uses `PELT` to compute a list of changing time points for each channel (Line 3). Then t^* is used to store the earliest changing time points for each node (Line 4-8). After that, we remove the edge (u, v) from G if the sudden MDR change of u does not lead to the MDR change of v (Line 9-11). Rocas



(a) Channel Module Graph (b) Message definition of the Planning channel (c) Tree representation for a message

Figure 7: How Rocas represents one execution. In (a), yellow ovals denote modules and blue boxes denote channels.

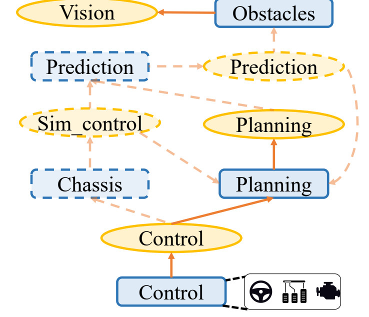


Figure 8: Reversed CMG in Alg. 1.

Algorithm 1: Pinpoint initial deviating module

1 **Function** `Find_Init_Module` *(G, channel2mdr)*
   // *channel2mdr* : map from channels to MDR series
   // t^* : map from node to its earliest changing point
2 $t^* = \emptyset$
3 *channel2chng_pt* = `PELT`(*channel2mdr*)
4 **for** *node* ∈ *G.nodes* **do**
5   **if** *node is Channel* **then**
6   t^*[node] = *channel2chng_pt*[node][0]
7   **if** *node is Module* **then**
8     $t^*[\text{node}] = \min_{(node,ch) \in G.edges} t^*[\text{ch}]$
9 **for** *(u,v)* ∈ *G.edges* **do**
10   **if** $t^*(u) \notin [t^*(v) - \delta, t^*(v)]$ **then**
11   G.`remove_edge`(u, v)
12 *ctrlnode* = `Control` channel node
13 $D = \{\, m \mid m$ is Module and reachable from *ctrlnode* in reversed $G\,\}$
14 **return** $argmin_{m \in D}\ t^*(m)$

use $\delta = 3$ because 3 seconds is a common reaction time in driving. Finally, starting from *ctrlnode*, the algorithm traverses from the reversed CMG *revG*, shown in Fig. 8, and stores all reachable nodes in set D (Line 12-14). The algorithm returns the module in D that has the earliest time changing point of MDR, namely the initial deviating module.

5.3 Cyber Mutation

After Alg. 1, we have obtained the initial deviating module. As the final phase to localize the accident-inducing cyber parameter setting, Rocas aims to find a minimal mutation on the parameter space within the identified module so that the original accident can be suppressed. We use a similar approach as Section 5.1. Note that the difference from Section 5.1 is that Rocas only mutates the internal cyber parameters in this phase.

The main motivation for modifying multiple configurations is that modifying only one configuration per iteration is much slower. In practice, the majority of the thousands of ADS configurations are irrelevant to the accident. Modifying multiple configurations simultaneously is more likely to hit the responsible configurations in a shorter time.

Our fitness function requires (1) the replay execution yields false. (i.e., accident suppressed); (2) the ADS's trajectory after mutation should be similar to the original one; (3) the set of mutated physical configurations is small; and (4) their value changes are small.

Given an initial cyber configuration $c_0 \in C$ such that $E(c_0) = 1$, finding the misconfiguration (i.e., minimum mutation) can be formulated as:

$$\begin{aligned} &\textbf{minimize } \mathcal{H}(c) = \{h_1(c), h_2(c)\}, \\ &\textbf{subject to } \mathcal{K}(c) = \{E(c) = 0\} \end{aligned} \tag{5}$$

$\mathcal{H}$ represents the function we aim to minimize, while $\mathcal{K}$ denotes the constraint that must be satisfied. $h_1(\cdot)$ defined in Eq. 6 measures the trajectory similarity (computed by MSE), and $Traj_c$ denotes the trajectory of ADS with the cyber configuration c. This objective function is necessary, otherwise one can just set the ADS's speed to zero and the accident can be suppressed.

$$h_1(c) = -Sim(Traj_{c_0}, Traj_c) \tag{6}$$

$h_2(\cdot)$ has the same purpose as $f_3(\cdot)$ in Eq. 4. We do not repeat details here due to space concerns.

$$h_2(c) = \#(c - c_0) + \sum_{k \in Enum} L_1(k_0, k) + \sum_{k \notin Enum} L_2(k_0, k) \tag{7}$$

6 EVALUATION

We introduce our evaluation setup (Section 6.1) and present results by answering the following research questions (RQs):
RQ1: How effective is Rocas on finding triggering entities and misconfigurations? (Section 6.2)
RQ2: How effective is the proposed metric MDR? (Section 6.3)
RQ3: How efficient is Rocas and each phase? (Section 6.4)
RQ4: How does AD root cause analysis help the accident investigation? (Case studies in Section 6.5)

6.1 Setup

Implementation. Rocas includes several components: (1) The replay engine that can replay actor trajectories inside simulator; (2) The physical mutator that can change the behaviors of actor entities and map entities; (3) The graph extractor that dynamically extracts CMG. (4) The execution differentiation algorithm that identifies the initial deviating module; (5) The cyber mutator that pinpoints the misconfiguration of the subject ADS. We run experiments on Ubuntu 20.04, with 96 GB RAM and Nvidia GPU 3090.
Subject Systems. We evaluate Rocas on two open-source ADSs, Baidu Apollo [2] and LAV [21], both designed in a modularized paradigm. Apollo is representative for industry-grade ADS, as Baidu has obtained permits to operate fully autonomous taxis without any human assistance in China since August 2022. LAV is a research-oriented ADS. We use it to show that Rocas can be generalized to

Table 2: Accident Types Summary. Fig. 14 illustrates scenarios. Link [14] shows accident videos.

Type	Src.†	#Inst.	SUT	Description	Conseq.	Scenario
A1	CA/DF/DT	37	Apollo 7.0	Incorrectly takes a slowly moving car as a static object	Collision	Intersection
A2	PF	3	Apollo 7.0	Blocked by 2 stopped vehicles ahead although there is enough space in between	EB	Following
A3	CA/AT	3	Apollo 7.0	Turns left at junction and collides with a truck from opposite direction when trying to overtake it	Collision	Intersection
A4	AT/DF/DT	43	Apollo 7.0	Over-aggressively overtakes a left-turning vehicle	Collision	Merging
A5	CA/AT	36	Apollo 7.0	Oscillates between overtake and yield and cannot stop in time when finally deciding to yield	Collision	Intersection
A6	DF	2	Apollo 7.0	Fails to avoid collision with a large truck drifting on wet ground	Collision	Intersection
A7	DF	4	Apollo 7.0	Collides with other vehicles from lateral or rear direction	Collision	Merging
A8	DF	3	LAV	Collides with road-side curbstone or vegetation	Collision	Following
A9	DF/DT	5	LAV	Takes a red truck as red light and suddenly stops in roundabout	EB	Intersection
A10	DF	2	LAV	Recognizes the wrong traffic light due to special road shape and suddenly stops on the road	EB	Turning
A11	DF	2	LAV	Turns right and fails to avoid collision with a wide truck	Collision	Merging
A12	DF/PF	4	LAV	Predicts the wrong trajectories of a cyclist on the side of AD vehicle	EB	Turning

Src: Accident Source, #Inst.: Instance Number, SUT: System Under Test, Conseq.: Consequences, EB: Emergency Braking.

† Accident Sources. CA: CAT [71], AT: ATLAS [73], DF: DriveFuzz [52], DT: DoppelTest [42], PF: PlanFuzz [77].

other modularized ADS, even if it is not based on ROS-like communication frameworks. Specifically, we manually isolate different modules by inspecting the code and instrument the data transmitted between modules. Such data approximates the channel messages in an ROS-like framework. The manual work is a one-time effort and affordable (2 man-hours). For simulation, we use the simulator supported by the corresponding ADS, namely LGSVL [5] for Apollo and CARLA [4] for LAV.

Logging and Replay. During ADS's execution, it is crucial that Rocas records the information of other actor entities. During the post-accident replay, Rocas needs to extract the locations and rotations of each actor entity from the recorded messages. Since these values are usually in the AD system's coordinates, Rocas transforms them into the simulator's world coordinates, using the transformation matrices of respective simulators.

Accident Cases. We investigate 184 accident cases from recent literature [42, 52, 71–73, 77]. These cases had been confirmed and hence can serve as the ground truth. We consider our tool correctly identifies a root cause if it locates the same misconfiguration. We categorize them via 3 perspectives: ❶ whether it can be exploited by leveraging external scenario , ❷ whether the accident consequence is severe (i.e., collision or emergency braking) or just efficiency degradation (e.g., taking longer routes), ❸ what the driving scenario is at accident moment. Detailed statistics are shown in Table 7 (in Supplementary Material of [6]). In order to eliminate non-safety-related incidents, we filter out cases that are not exploitable and solely related to performance issues. After the filtering process, as shown in Table 5 (in Supplementary Material of [6]), 144 cases are left, which we further categorize into 12 types based on similarities in ADS's behaviors, accident consequences, and driving scenarios. Table 2 shows a comprehensive overview of these categorized cases.

6.2 Effectiveness of Root Cause Analysis

Accident Root Cause Analysis. Table 4 consists of several columns that provide essential information regarding the accidents and the

Table 3: Scope reduction of configurations in Baidu Apollo.

Main Modules	# Config.	Proportion
Perception	171	15.35%
Prediction	183	16.53%
Planning	222	19.93%
Control	55	4.94%
All Modules	1114	100%

corresponding investigation results from each phase. There are three main categories based on the identified deviating module. (1) Prediction related accidents. In A1, the ADS incorrectly predicts a moving car as static, and in A10 the ADS predicts the wrong trajectory of a right-side cyclist. (2) Planning related accidents. A2, A6, and A11 are different scenarios where the ADS has the wrong decision on whether other entities have overlap with the ADS's planned trajectory. A3, A4, and A5 are different scenarios where the ADS has the wrong decision on whether it should overtake or yield to other entities. (3) Perception related accidents. In A9, the ADS fluctuates the braking probability when seeing a red truck ahead. In A12, due to the special road shape, the ADS recognizes the traffic light from the wrong direction. We further illustrate the accident scenarios for the cases in Table 2 in Figure 14 in the Supplementary Material of [6] (A7 in Table 2 has been illustrated in Figure 4).

Regression Test. We conduct regression tests to ensure that our mutated configurations do not introduce new hazardous behaviors. We constructed a regression dataset with 200 cases by adding small perturbations to the original executions. In our evaluation, all modified configurations achieved higher mission success rates than the original configurations.

Failure Cases. Rocas has certain limitations when dealing with two specific accident types, namely Type A7 and A8. Type A7 involves collisions with other vehicles approaching the stationary ADS from lateral or rear directions. In such cases, Rocas can identify the triggering entity during the physical mutation phase but is unable to modify a configuration value in the cyber mutation phase

Table 4: Root Cause Analysis Results. Physical Mutation 4 columns, display Phase-I output (i.e., the mutated value of entity properties v.s. original values). Deviating Module column shows Phase-II output. Cyber Mutation 3 columns, display Phase-III output (i.e., the mutated value of configurations v.s. original values).

Type	System	Physical Mutation				Deviating Module	Cyber Mutation		
		Triggering Entity	Property	Ori. Val.	Mut. Val.		Misconfiguration	Ori. Val.	Mut. Val.
A1	Apollo 7.0	Collided Sedan	Speed	0.98	1.2	Prediction	`still_obstacle_speed_threshold`	0.99	0.50
A2	Apollo 7.0	Right-front Sedan	Location.x	14.42	14.37	Planning	`obstacle_lat_buffer`	0.60	0.20
A3	Apollo 7.0	Collided Truck	Rotation.yaw Location.y	275 -27.1	285 -27.3	Planning	`kMinOvertakeDistance`	10.0	40.0
A4	Apollo 7.0	Collided Sedan	Speed	6.0	4.0	Planning	`yield_distance`	5.0	2.0
A5	Apollo 7.0	Collided Sedan	Location.x	-234.5	-235.2	Planning	`kOvertakeTimeBuffer`	3.0	2.0
A6	Apollo 7.0	Collided Truck	Rotation.yaw	310	300	Planning	`kADCSafetyLBuffer`	0.1	1.0
A7	Apollo 7.0	Collided Sedan	Location.x	13.5	13.0	Perception	-	-	-
A8	LAV	Collided Building	-	Enabled	Disabled	Perception	-	-	-
A9	LAV	Front-side truck	–	Enabled	Disabled	Perception	`brake_threshold`	0.10	0.20
A10	LAV	Left-side Traffic Light	Policy	Red	Green	Perception	`brake_threshold`	0.10	0.90
A11	LAV	Left-front Truck	Rotation.yaw	32	25	Planning	`dist_threshold_moving`	2.50	3.50
A12	LAV	Right-side cyclist	Speed	1.0	0.0	Prediction	`dist_threshold_static`	1.00	0.50

to prevent the accident. This limitation arises because most ADSs do not incorporate features to prevent collisions from lateral or rear directions, as such collisions are generally considered inevitable and not within the typical expectations for ADSs to handle. As for Type A8, it pertains to collisions with road-side curbstones or vegetation that occur when the ADS follows abnormal trajectories. Upon manual investigation, we found that these collisions occur when there is no feasible path from the current AD's location to the routing destination. As a result, the ADS randomly deviates from its original trajectories. This situation may arise when the AD vehicle is spawned at a junction or roads with opposite directions, or when the AD vehicle is blocked at certain corners where the ADS lacks a back-up feature to navigate out of the situation. In summary, the collision incidents in Type A7 and A8 are primarily linked to the lack of feature implementation, such as the absence of measures to avoid lateral collisions or handle infeasible routing, rather than being caused by misconfigurations.

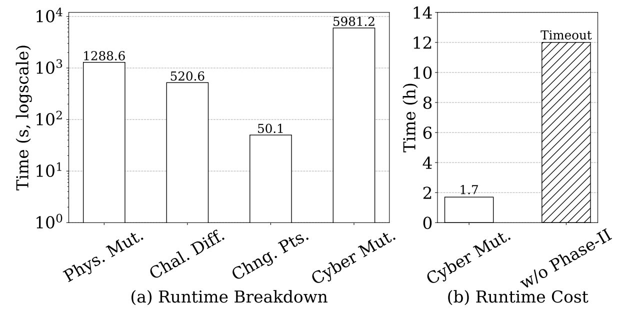


Figure 9: Runtime. (a) shows runtime of each phase. (b) shows the benefit of search scope reduction.

below 20% of the original whole search space. This significant reduction in search space contributes to the runtime efficiency of Phase-III (cyber mutation). We also include a concrete example in Supplementary Material of [6] Section A to illustrate how Rocas narrows down the search space.

6.3 Effectiveness of MDR

Table 3 shows Rocas in average reduces search space of configurations by 85.8%. Table 3 provides a detailed breakdown of the number of configurable parameters in Apollo's four main modules, including the perception, prediction, planning, and control module (Rows 2-5). The sum of configurable parameters across all modules is also presented in Row 6. We can see that, due to the complexity of the Apollo system, there are a significant number of configurable parameters, which can lead to misconfigurations and subsequent failures. However, Rocas has proven to be an effective tool for narrowing down the search space and identifying these misconfigurations. By leveraging its execution diff algorithm, Rocas successfully reduces the misconfiguration search space to

6.4 Efficiency of Runtime

Because the length of an execution can impact the investigation runtime (for instance, a longer execution can result in a larger mutation search space and more execution records to analyze), we trim executions to the last 30 seconds before the occurrence of accidents. By doing so, we can limit the amount of data we need to analyze while still capturing the critical period leading up to the accidents. The average storage size for a 30-second execution is around 30 MB. The experimental results show that Phase-III (i.e., the cyber mutation) represents 76.3% of the total time. It demonstrates that the scope reduction of Phase-II is necessary to make the root cause analysis affordable. On the other hand, a baseline method without scope reduction cannot find any misconfiguration within a time

budget of 12 hours. Details are discussed below.

Time Breakdown. Figure 9a displays the average runtime cost for each step during an accident investigation. Phase-I (physical mutation) takes up 16.4% of the total runtime, while Phase-II (including channel message differential analysis and change point detection) accounts for 7.3%. The most time-consuming part is Phase-III (cyber mutation) which takes up 76.3% of the total time.

The reason why Phase-I is much faster than Phase-III is that it is easier to identify a mutation that can prevent accidents when modifying physical entities (e.g., changing the location or speed of other vehicles). However, in Phase-III, Rocas must identify the responsible mis-configuration among many possible parameters to avoid the accident. Therefore, the scope reduction of Phase-II is necessary to make the root cause analysis affordable.

Compare with Baseline. Since there is no existing work that can be directly applied on the ADS root cause analysis task, we compare Rocas with a naive brute force method that searches among all ADS modules. Figure 9b shows the average runtime comparison for cyber mutation, with and without the scope reduction. The naive brute force method cannot find a misconfiguration within a reasonable budget (12 hours).

6.5 Case Studies

We present the details of two cases (A4 and A6 in Table 2) to demonstrate the benefits of Rocas and how the found misconfiguration can shed light on the mystery behind the accidents. Case Study I and II are in Supplementary Material of [6] Section B.

7 DISCUSSIONS

Practicality. We use simulated accidents instead of real-world accidents for the following reasons. Firstly, accidents data of real-world deployed ADS are typically considered proprietary by companies and not publicly available. Secondly, conducting real-world experiments on accident investigation is often prohibitively expensive. Additionally, recent advances in high-fidelity simulators [4, 5, 8] enable simulations to closely reflect real-world conditions.

Limitations. Rocas is limited to modularized ADS and cannot be applied to the end-to-end (e.g., reinforcement learning) ADS. In the latter case, Rocas can only identify the triggering entity, but it cannot pinpoint the specific module or misconfiguration.

Validation. Rocas outputs a misconfiguration, but it still requires human to inspect misconfiguration usages to finally understand the rationale behind accidents. Also, Rocas can help *finetune*, but not *fix* misconfigurations of ADS. Fixing configuration issues is complimentary to Rocas and still an open challenging problems in SE community [26, 33, 36, 37, 39, 57, 74, 75]. We leave it as future work.

8 RELATED WORK

CPS Fuzzing. Inspired by traditional fuzzing methods [1, 16, 63, 81], recently many CPS fuzzing techniques have been proposed to address unique challenges in CPS including drone systems [35, 50, 54], ADS [28, 29, 43, 52, 69, 77, 84, 88, 89], ROS [51, 80], DNN controllers [45], and other systems [19, 61, 64, 66]. Our post-accident analysis is complementary to the fuzzing methods mentioned above.

Root Cause Analysis. Root cause analysis is critical for debugging program failures. Numerous techniques have been proposed, such as log-based causality analysis [31, 55, 56, 58, 62]. Some use program instrumentation to generate execution logs [68, 83], while others focus on recording OS events during runtime and performing offline analysis. Techniques like taint analysis [17, 25, 47] and record-and-replay provenance analysis [20, 30–32, 55, 67, 70, 76] are also widely used for accident investigation. However, these approaches induce heavy overhead, and are not suitable for resource-constrained CPS.

Fault localization. Fault localization techniques [79] vary widely, including slice-based [78], spectrum-based [38], statistics-based [87], states-based [85], learning-based [18], etc. Our method differs from existing fault localization methods in two aspects: (1) Fault localization methods typically pinpoint suspicious code element (e.g., statements, predicates, functions or files), while our method pinpoints the misconfiguration. (2) Fault localization methods focus on program bugs, while our focus is on misconfiguration, which does not necessarily mean that the code logic is buggy.

CPS Root Cause Analysis. Root cause analysis of CPS is a less investigated area. There are existing efforts focusing on drone systems [22, 24, 44, 53]. They either fail to capture complex behaviors across multiple modules in ADS [22, 24, 44] or requires heavy-weight program instrumentation and trace collection [53]. A recent work CARE [40] focuses on mission failures in robots, it cannot reason environments with moving objects. On the other hand, Swarmbug [46] focuses on problems in the swarm coordination, not considering internal logic of individual agents. Compared with the above techniques, our work is the first to target at AD scenarios that can handle both more complex physical environments and internal program configurations.

9 CONCLUSION

In this paper, we first formally define the problem of ADS root cause analysis and present Rocas, a novel method for ADS root cause analysis. Our technique leverages both physical and cyber mutation that can precisely identify the accident-trigger entity and pinpoint the misconfiguration responsible for an accident. We demonstrate the effectiveness and efficiency of Rocas through the evaluation of 12 types of ADS accidents, 144 accidents in total.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments and suggestions. We are grateful to the Center for AI Safety for providing computational resources. This work was funded in part by the National Science Foundation (NSF) Awards SHF-1901242, SHF-1910300, IIS-2416835, DARPA VSPELLS - HR001120S0058, IARPA TrojAI W911NF-19-S0012, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] [n. d.]. *American Fuzzy Lop.* https://lcamtuf.coredump.cx/afl/

[2] [n. d.]. *Baidu Apollo.* https://github.com/ApolloAuto/apollo

[3] [n. d.]. *Baidu Apollo CyberRT.* https://cyber-rt.readthedocs.io/en/latest/

[4] [n. d.]. *CARLA simulator.* https://carla.org/

[5] [n. d.]. *LGSVL simulator.* https://www.svlsimulator.com/

[6] [n. d.]. *ROCAS Artifact Repo.* https://github.com/GiantSeaweed/ROCAS

[7] [n. d.]. *TESLA KEEPS "SLAMMING ON THE BRAKES" WHEN IT SEES STOP SIGN ON BILLBOARD.* https://futurism.com/the-byte/tesla-slamming-brakes-sees-stop-sign-billboard

[8] [n. d.]. *Waymo. 2020. Off road, but not offline: How simulation helps advance our Waymo Driver.* https://blog.waymo.com/2020/04/off-road-but-not-offline--simulation27.html

[9] 2016. *BBC News. 2016. Google Self-driving Car Hits a Bus.* https://www.bbc.com/news/technology-35692845

[10] 2016. *BBC News. 2016. Uber in Fatal Crash Had Safety Flaws Say US Investigators.* https://www.bbc.com/news/business-50312340

[11] 2019. *BBC News. 2019. Tesla Model 3: Autopilot Engaged during Fatal Crash.* https://www.bbc.com/news/technology-48308852

[12] 2020. *Associated Press News. 2020. 3 Crashes, 3 Deaths Raise Questions About Tesla's Autopilot.* https://apnews.com/article/europe-us-news-ap-top-news-in-state-wire-mi-state-wire-ca5e62255bb87bf1b151f9bf075aaadf

[13] 2021. CARLA AD Challenge. https://leaderboard.carla.org/.

[14] 2023. *Accident Videos.* https://www.youtube.com/playlist?list=PLagazsxgy9BD2SdAO66ZR-rmfimeWpsgF

[15] 2023. *WIRED. 2023. Dashcam Footage Shows Driverless Cars Clogging San Francisco.* https://www.wired.com/story/dashcam-footage-shows-driverless-cars-cruise-waymo-clogging-san-francisco/

[16] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[17] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The world's fastest taint tracker. In *Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings 14*. Springer, 1–20.

[18] Yuriy Brun and Michael D Ernst. 2004. Finding latent code errors via machine learning over program executions. In *Proceedings. 26th International Conference on Software Engineering*. IEEE, 480–490.

[19] Giuseppe Cascavilla, Johann Slabber, Fabio Palomba, Dario Di Nucci, Damian A. Tamburri, and Willem-Jan van den Heuvel. 2020. Counterterrorism for Cyber-Physical Spaces: A Computer Vision Approach. In *Proceedings of the International Conference on Advanced Visual Interfaces* (Salerno, Italy) *(AVI '20)*. Association for Computing Machinery, New York, NY, USA, Article 52, 5 pages. https://doi.org/10.1145/3399715.3399826

[20] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. 2011. Intrusion recovery for database-backed web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 101–114.

[21] Dian Chen and Philipp Krähenbühl. 2022. Learning from all vehicles. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 17222–17231.

[22] Hongjun Choi, Zhiyuan Cheng, and Xiangyu Zhang. 2022. RVPLAYER: Robotic Vehicle Forensics by Replay with What-if Reasoning. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-215/

[23] Hongjun Choi, Sayali Kate, Yousra Aafer, Xiangyu Zhang, and Dongyan Xu. 2020. Cyber-Physical Inconsistency Vulnerability Identification for Safety Checks in Robotic Vehicles. https://doi.org/10.1145/3372297.3417249

[24] Devon R Clark, Christopher Meffert, Ibrahim Baggili, and Frank Breitinger. 2017. DROP (DRone Open source Parser) your drone: Forensic analysis of the DJI Phantom III. *Digital Investigation* 22 (2017), S3–S14.

[25] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 196–206.

[26] Raphael Pereira de Oliveira, Paulo Anselmo da Mota Silveira Neto, Qi Hong Chen, Eduardo Santana de Almeida, and Iftekhar Ahmed. 2022. Different, Really! A comparison of Highly-Configurable Systems and Single Systems. *Information and Software Technology* 152 (2022), 107035.

[27] Kalyanmoy Deb and Deb Kalyanmoy. 2001. *Multi-Objective Optimization Using Evolutionary Algorithms*. USA.

[28] Hina Anwar Dietmar Pfahl Fauzia Khan, Laima Dalbina. 2023. How Can Simulation-based Safety Testing Help Understand the Real-World Safety of Autonomous Driving Systems? *WORKS in PROGRESS in EMBEDDED COMPUTING (WiPiEC)* (2023).

[29] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, Chen, and Qi Alfred. 2020. A Comprehensive Study of Autonomous Vehicle Bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 385–396. https://doi.org/10.1145/3377811.3380397

[30] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2007. Friday: Global Comprehension for Distributed Replay.. In *NSDI*, Vol. 7. 285–298.

[31] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. 2005. The taser intrusion recovery system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 163–176.

[32] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay.. In *OSDI*, Vol. 8. 193–208.

[33] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1095–1106. https://doi.org/10.1109/ICSE.2019.00113

[34] Gor Hakobyan and Bin Yang. 2019. High-performance automotive radar: A review of signal processing algorithms and modulation schemes. *IEEE Signal Processing Magazine* 36, 5 (2019), 32–44.

[35] Ruidong Han, Chao Yang, Siqi Ma, JiangFeng Ma, Cong Sun, Juanru Li, and Elisa Bertino. 2022. Control parameters considered harmful: Detecting range specification bugs in drone configuration modules via learning-guided search. In *Proceedings of the 44th International Conference on Software Engineering*. 462–473.

[36] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Ciudad Real, Spain) *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 23, 10 pages. https://doi.org/10.1145/2961111.2962602

[37] Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 17–28. https://doi.org/10.1145/3238147.3238204

[38] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. 2000. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability* 10, 3 (2000), 171–194.

[39] Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. 2020. CP-Detector: Using Configuration-related Performance Properties to Expose Performance Bugs. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 623–634.

[40] Md Abir Hossen, Sonam Kharade, Bradley Schmerl, Javier Cámara, Jason M O'Kane, Ellen C Czaplinski, Katherine A Dzurilla, David Garlan, and Pooyan Jamshidi. 2023. CaRE: Finding Root Causes of Configuration Issues in Highly-Configurable Robots. *IEEE Robotics and Automation Letters* (2023).

[41] Daniel Howard and Danielle Dai. 2014. Public perceptions of self-driving cars: The case of Berkeley, California. In *Transportation research board 93rd annual meeting*, Vol. 14. The National Academies of Sciences, Engineering, and Medicine Washington, DC, 1–16.

[42] Yuqi Huai, Yuntianyi Chen, Sumaya Almanee, Tuan Ngo, Xiang Liao, Ziwen Wan, Qi Alfred Chen, and Joshua Garcia. 2023. Doppelganger Test Generation for Revealing Bugs in Autonomous Driving Software. In *ACM/IEEE 45th International Conference on Software Engineering*. Melbourne, Australia.

[43] Dmytro Humeniuk, Foutse Khomh, and Giuliano Antoniol. 2022. A search-based framework for automatic generation of testing environments for cyber-physical systems. *Information and Software Technology* 149 (2022), 106936.

[44] Upasita Jain, Marcus Rogers, and Eric T Matson. 2017. Drone forensic framework: Sensor and data identification and verification. In *2017 IEEE Sensors Applications Symposium (SAS)*. IEEE, 1–6.

[45] Chijung Jung, Ali Ahad, Yuseok Jeon, and Yonghwi Kwon. 2022. SwarmFlawFinder: Discovering and Exploiting Logic Flaws of Swarm Algorithms. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, 23-26 May 2021*. IEEE, 1447–1464. https://doi.org/10.1109/SP46214.2022.00084

[46] Chijung Jung, Ali Ahad, Jinho Jung, Sebastian Elbaum, and Yonghwi Kwon. 2021. Swarmbug: Debugging Configuration Bugs in Swarm Robotics. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 868–880. https://doi.org/10.1145/3468264.3468601

[47] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. Dta++: dynamic taint analysis with targeted control-flow propagation.. In *NDSS*.

[48] Ranjita Pai Kasturi, Yiting Sun, Ruian Duan, Omar Alrawi, Ehsan Asdar, Victor Zhu, Yonghwi Kwon, and Brendan Saltaformaggio. 2020. TARDIS: Rolling back the clock on CMS-targeting cyber attacks. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1156–1171.

[49] R. Killick, P. Fearnhead, and I. A. Eckley. 2012. Optimal Detection of Changepoints With a Linear Computational Cost. *J. Amer. Statist. Assoc.* 107, 500 (oct 2012), 1590–1598. https://doi.org/10.1080/01621459.2012.737745

[50] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z Berkay Celik, and Dongyan Xu. 2021. PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

[51] Seulbae Kim and Taesoo Kim. 2022. RoboFuzz: Fuzzing Robotic Systems over Robot Operating System (ROS) for Finding Correctness Bugs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022).* Association for Computing Machinery, New York, NY, USA, 447–458. https://doi.org/10.1145/3540250.3549164

[52] Seulbae Kim, Major Liu, Junghwan "John" Rhee, Yeseok Jeon, Yonghwi Kwon, and Chung Hwan Kim. 2022. DriveFuzz: Discovering Autonomous Driving Bugs through Driving Quality-Guided Fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS 2022).* ACM, 1753–1767. https://doi.org/10.1145/3548606.3560558

[53] Taegyu Kim, Chung Hwan Kim, Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave (Jing) Tian, and Dongyan Xu. 2020. From Control Model to Program: Investigating Robotic Aerial Vehicle Accidents with MAYDAY. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 913–930. https://www.usenix.org/conference/usenixsecurity20/presentation/kim

[54] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFUZZER: Finding Input Validation Bugs in Robotic Vehicles through Control-Guided Testing. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19).* USENIX Association, USA, 425–442.

[55] Taesoo Kim, Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, et al. 2010. Intrusion Recovery Using Selective Re-execution.. In *OSDI*. 89–104.

[56] Samuel T King and Peter M Chen. 2003. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles.* 223–236.

[57] Rahul Krishna, Md Shahriar Iqbal, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2020. Cadet: Debugging and fixing misconfigurations using counterfactual reasoning. *arXiv preprint arXiv:2010.06061* (2020).

[58] Srinivas Krishnan, Kevin Z Snow, and Fabian Monrose. 2010. Trail of bytes: efficient support for forensic analysis. In *Proceedings of the 17th ACM conference on Computer and communications security.* 50–60.

[59] Bijun Lee, Yang Wei, and I Yuan Guo. 2017. Automatic parking of self-driving car based on lidar. *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci* 42 (2017), 241–246.

[60] Henry C Lee and Elaine M Pagliaro. 2013. Forensic evidence and crime scene investigation. *Journal of Forensic Investigation* 1, 2 (2013), 1–5.

[61] Jaekwon Lee, Enrico Viganò, Oscar Cornejo, Fabrizio Pastore, and Lionel Briand. 2023. Fuzzing for CPS Mutation Testing. arXiv:2308.07949 [cs.SE]

[62] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *NDSS*, Vol. 16.

[63] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. In *32nd USENIX Security Symposium (USENIX Security 23).* USENIX Association, Anaheim, CA, 1379–1396. https://www.usenix.org/conference/usenixsecurity23/presentation/li-wen (artifact evaluated; badges: Available).

[64] Prianka Mandal, Sunil Manandhar, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2023. Helion: Enabling Natural Testing of Smart Homes. *arXiv preprint arXiv:2308.06695* (2023).

[65] Rui Mei, Hanbing Yan, Qinqin Wang, Zhihui Han, and Zhuohang Lyu. 2022. TDLens: toward an empirical evaluation of provenance graph-based approach to cyber threat detection. *China Communications* 19, 10 (2022), 102–115.

[66] Claudio Menghi, Enrico Viganò, Domenico Bianculli, and Lionel C Briand. 2021. Trace-checking CPS properties: Bridging the cyber-physical gap. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 847–859.

[67] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2006. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro* 26, 1 (2006), 100–109.

[68] Peter Ohmann and Ben Liblit. 2017. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *Automated Software Engineering* 24 (2017), 865–904.

[69] Qi Pang, Yuanyuan Yuan, and Shuai Wang. 2022. MDPFuzz: Testing Models Solving Markov Decision Processes. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022).* Association for Computing Machinery, New York, NY, USA, 378–390. https://doi.org/10.1145/3533767.3534388

[70] Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, Yuanyuan Zhou, et al. 2004. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track.* Boston, MA, USA, 29–44.

[71] Yun Tang, Yuan Zhou, Yang Liu, Jun Sun, and Gang Wang. 2021. Collision Avoidance Testing for Autonomous Driving Systems on Complete Maps. In *2021 IEEE Intelligent Vehicles Symposium (IV).* 179–185. https://doi.org/10.1109/IV48863.2021.9575536

[72] Yun Tang, Yuan Zhou, Fenghua Wu, Yang Liu, Jun Sun, Wuling Huang, and Gang Wang. 2021. Route coverage testing for autonomous vehicles via map modeling. In *2021 IEEE International Conference on Robotics and Automation (ICRA).* IEEE, 11450–11456.

[73] Yun Tang, Yuan Zhou, Tianwei Zhang, Fenghua Wu, Yang Liu, and Gang Wang. 2021. Systematic Testing of Autonomous Driving Systems Using Map Topology-Based Scenario Classification. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1342–1346. https://doi.org/10.1109/ASE51524.2021.9678735

[74] Pablo Valle, Aitor Arrieta, and Maite Arratibel. 2023. Automated Misconfiguration Repair of Configurable Cyber-Physical Systems with Search: an Industrial Case Study on Elevator Dispatching Algorithms. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* IEEE. https://doi.org/10.1109/icse-seip58684.2023.00042

[75] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2022. On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support. In *Proceedings of the 44th International Conference on Software Engineering (ICSE).* ACM Press, New York, NY, 1571–1583. https://doi.org/10.1145/3510003.3510043

[76] Gregory Walkup, Sriharsha Etigowni, Dongyan Xu, Vincent Urias, and Han W Lin. 2020. Forensic Investigation of Industrial Control Systems Using Deterministic Replay. In *2020 IEEE Conference on Communications and Network Security (CNS).* IEEE, 1–9.

[77] Ziwen Wan, Junjie Shen, Jalen Chuang, Xin Xia, Joshua Garcia, Jiaqi Ma, and Qi Alfred Chen. 2022. Too Afraid to Drive: Systematic Discovery of Semantic DoS Vulnerability in Autonomous Driving Planning under Physical-World Attacks. In *Network and Distributed System Security (NDSS) Symposium, 2022.*

[78] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[79] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[80] Kai-Tao Xie, Jia-Ju Bai, Yong-Hao Zou, and Yu-Ping Wang. 2022. ROZZ: Property-based Fuzzing for Robotic Programs in ROS. In *2022 International Conference on Robotics and Automation (ICRA).* IEEE, 6786–6792.

[81] Xiaofei Xie, Hongxu Chen, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. Coverage-Guided Fuzzing for FeedForward Neural Networks. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1162–1165.

[82] Ibrar Yaqoob, Latif U Khan, SM Ahsan Kazmi, Muhammad Imran, Nadra Guizani, and Choong Seon Hong. 2019. Autonomous driving cars in smart cities: Recent advances, requirements, and challenges. *IEEE Network* 34, 1 (2019), 174–181.

[83] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems.* 143–154.

[84] Fiorella Zampetti, Ritu Kapur, Massimiliano Di Penta, and Sebastiano Panichella. 2022. An empirical characterization of software bugs in open-source Cyber-Physical Systems. *J. Syst. Softw.* 192 (2022), 111425. https://doi.org/10.1016/j.jss.2022.111425

[85] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Charleston, South Carolina, USA) *(SIGSOFT '02/FSE-10).* Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/587051.587053

[86] Jun Zeng, Chuqi Zhang, and Zhenkai Liang. 2022. PalanTír: Optimizing Attack Provenance with Hardware-enhanced System Observability. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.* 3135–3149.

[87] Alice X Zheng, Michael I Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning.* 1105–1112.

[88] Ziyuan Zhong, Zhisheng Hu, Shengjian Guo, Xinyang Zhang, Zhenyu Zhong, and Baishakhi Ray. 2022. Detecting multi-sensor fusion errors in advanced driver-assistance systems. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis.* 493–505.

[89] Ziyuan Zhong, Yun Tang, Yuan Zhou, Vania de Oliveira Neves, Yang Liu, and Baishakhi Ray. 2021. A Survey on Scenario-Based Testing for Automated Driving Systems in High-Fidelity Simulation. arXiv:2112.00964 [cs.SE]

SUPPLEMENTARY MATERIAL

A HOW ROCAS REDUCES SEARCH SPACE

As we mentioned in Section 5.2, due to the natural noise of the physical environment, there always exist some differences when conducting the differential analysis (in Section 5.2.3). Therefore it is necessary and crucial to locate the accident-inducing difference. Figure 16 shows the MDR series of different channels (for accident A1), corresponding to the CMG in Figure 15. Red dot lines denote the change points. This accident happens at 7.1s, marked as the red region in the `control` channel subfigure.

It is worth noting that different channels have different *default* MDR values. For example, the `planning` channel has a default MDR of 12.84, while `prediction` channel has a default MDR of 28.86, as shown in Table 6. When running Algorithm 1 (with $\delta = 3$) to find the deviating module, we can obtain a path in reversed CMG, namely `control` → `planning` → `prediction`. Therefore module `prediction` is correctly identified. Although channel `chassis` also has a change point at 1.5s, it is not reachable from channel `control` because the edge from channel `chassis` to module `control` has been removed at Algorithm 1 Line 11.

B CASE STUDIES

Case Study I: Yield Distance (Accident A4). In A4 (as shown in Figure 14c), the subject ADS (in blue) is driving ahead and intends to pass a crossing. At the same time, another vehicle (in red) is attempting a left turn into the same lane as the ADS. Unexpectedly, the ADS fails to yield to the red vehicle and collides with it.

To investigate the root cause of this accident, RoCAS first mutates the surrounding entities (including the red vehicle) to suppress the accident. RoCAS finds that the collision is prevented when the speed of the red vehicle is reduced from 6 m/s to 4 m/s. Next, RoCAS conducts the differential analysis on the executions with and without the accident and pinpoint to the planning module. RoCAS then mutates configurations within the planning module and finds that the collision can be suppressed by changing the value of `yield_distance`. Figure 10 illustrates different execution results with different `yield_distance` values. In the original execution with the accident, `yield_distance` is set to 5 (Figure 10a), and the collision still occurs when the value is increased to 10 (Figure 10b). However, the collision is avoided when the `yield_distance` is decreased to 2 (Figure 10c). It indicates that this vulnerability is related to the values of `yield_distance`.

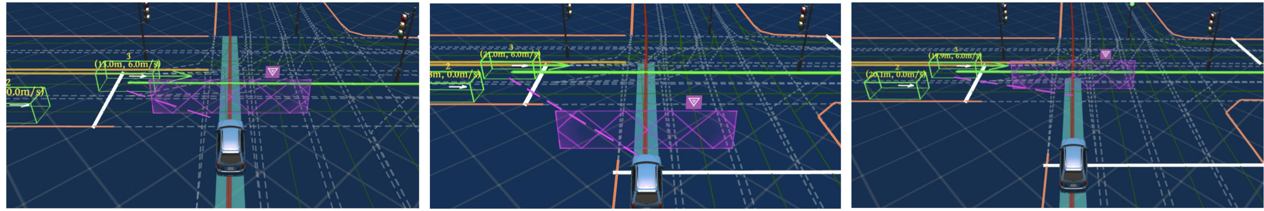
To further investigate the root case, we locate the usage of `yield_distance` in source code. Figure 11 shows the code snippet of `CreateYieldDecision` API, which is used to make the yield decision. In Line 4, parameter `yield_distance` is used to compute the location point at which the AD vehicle plans to stop for yielding ahead vehicles (e.g., the red one). When the value is too large, such as in Figure 10a and 10b, the stop location is too close for the subject AD vehicle to make a full stop. This causes the vehicle to make a plan of not yielding and thus triggers the collision. This vulnerability can be fixed by setting the parameter `yield_distance` to a smaller value, such as 2 used in Figure 10c.

Case Study II: Drifting Truck (Accident A6). Accident A6 (Figure 14e) depicts a more complicated accident case. As shown in the simulator screenshot, this accident occurs on a rainy day. The AD vehicle is driving ahead and passing a crossing when an external truck (in red) is turning left at the intersection. However, the ADS fails to yield the truck and collides with it.

To identify the vulnerability that causes this accident, RoCAS first replays the accident in the simulator and performs the physical mutation. By mutating the property `Rotation.yaw` from 310 to 300, RoCAS is able to find an accident-free execution. `Rotation.yaw` represents the angle between the ahead truck and the path that the ADS is traveling. Therefore, when the position of the truck is rotated to be outside the ADS's path, the collision can be avoided. Given this accident-free execution, RoCAS compares it with the accident execution that results in the accident and localizes the planning module as the problematic module. RoCAS then performs the cyber mutation on the configuration space within the planning module and finds that the accident can be suppressed by increasing the configuration `kADCSafetyLBuffer` from 0.1 to 1.

To investigate the root cause of the vulnerability, we examine the usage of `kADCSafetyLBuffer` in the source code. As shown in Figure 13, `kADCSafetyLBuffer` is used in the API `ComputeObstacleSTBoundary` (Line 5), which computes the boundary of external obstacles. Specifically, it computes the obstacle's bounding box based on its trajectory (Line 6 and Line 8) and determines whether the ADS will overlap with the obstacle's bounding box `obs_box` (Line 10). The configurable parameter `kADCSafetyLBuffer` is used to extend the overlap detection area by adding a buffer to its width for increased safety. A larger `kADCSafetyLBuffer` means the ADS detects external obstacles in a wider range. While we find that increasing `kADCSafetyLBuffer` can prevent the accident in this case, it may not be an optimal solution, as it could potentially trigger other vulnerabilities in different scenarios, such as making an unexpected stop or yield for an external obstacle with a safe distance.

Note that `kADCSafetyLBuffer` is used for additional safety consideration. Even when its value is set to zero, the collision should not occur as long as the obstacle is outside the ADS's path. This suggests that the vulnerability is likely caused by using an incorrect obstacle bounding box, i.e., `obs_box`, another variable considered in overlap checking (Line 10). The results of physical mutation also confirm this hypothesis, where RoCAS finds that adjusting the truck's direction can avoid the accident. `obs_box` is obtained by calling `GetBoundingBox` (Line 8), defined at Line 2. Upon investigation, we find that a bounding box contains information such as the position (x and y in Line 3) and the angle (theta in Line 4). However, the code uses the tangent to the path as the angle (i.e., `path_point().theta()` in Line 4), instead of the real angle of the truck. We illustrate the difference between the two values in Figure 12a. The red curve shows the trajectory of the truck, and the red box in dotted line shows the computed bounding box by the ADS, which has no overlap with the path of the ADS. However, due to the rainy weather, the large truck drifts on wet ground during the turn, making its real angle different from the tangent to the path. Thus, the real bounding box (the black box) is different from the computed one, which actually overlaps with the path of the ADS. It indicates that the vulnerability can be fixed by replacing



(a) yd=5. Collision. (b) yd=10. Collision. (c) yd=2.No collision.

Figure 10: Replay Accident A4 with different values for `yield_distance` (yd for short)

```
1 const double yield_distance = 5.0
2 bool SpeedDecider::CreateYieldDecision(Obstacle& obstacle,
3   ObjectDecisionType* yield_decision) {
4     yield_decision->set_distance_s(-yield_distance); // set YIELD decision
5     ... }
```

Figure 11: Simplified code logic for Accident A4.

(a) Illustration. (b) `Path.theta`. v = 10 km/h. Collision. (c) `Obs.theta`. v= 0 km/h. No collision.

Figure 12: Replay Accident A6. In Figure 12a, black box is the truck. Red dotted box is what AD vehicle thinks.

```
1 const double kADCSafetyLBuffer = 0.1
2 Box2d Obstacle::GetBoundingBox(TrajectoryPoint& point) const {
3   return Box2d(point.path_point().x(), point.path_point().y(),
4               point.path_point().theta(), ...); }
5 bool ComputeObstacleSTBoundary(Obstacle& obs, ...) {
6   auto& obs_traj = obstacle.Trajectory();
7   for (auto& obs_tjpt : obs_traj.traj_point()) {
8     Box2d& obs_box = obs.GetBoundingBox(obs_tjpt);
9     // adc_path_points is AD Car's planning path
10    if (GetOverlappingS(adc_path_points, obs_box, kADCSafetyLBuffer)) {...}
11 } }
```

Figure 13: Simplified code logic for Accident A6

Table 5: Selecting Criteria. From Table 7 we select exploitable and severe cases (i.e., collision or emergency braking).

	Inefficient	Severe	All
Exploitable	13	**144**	157
Inexploitable	12	15	27
All	25	159	184

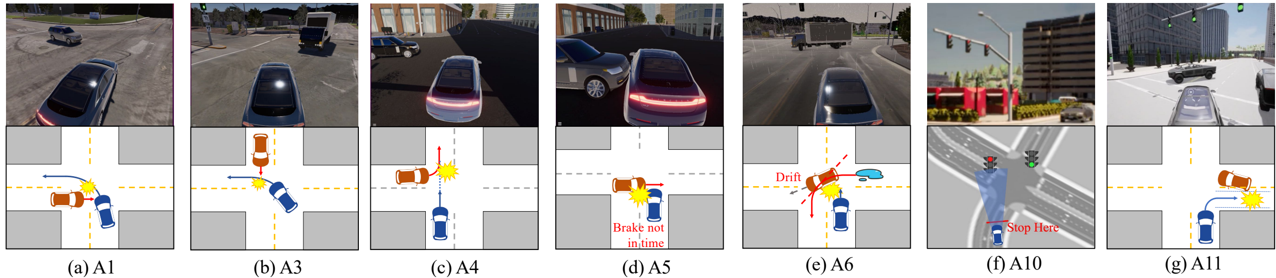
`path_point().theta()` with `obs.theta()`. When using `path_point().theta()` in the original execution (Figure 12b), the ADS does not yield and collides with the truck at a velocity of 10 km/h. However, when using `obs.theta()` (Figure 12c), the ADS detects the truck successfully and stops completely, waiting for the truck to bypass.

Table 6: Change time point t^* for each channel and the MDR value before and after MDR.

Channel Name	$t*$	MDR Before	MDR After	MDR Δ Ratio
/apollo/perception/obstacles	-	-	-	-
/apollo/prediction	4.50	28.86	30.72	+1.86
/apollo/planning	7.00	12.84	17.44	+4.60
/apollo/canbus/chassis	1.50	4.28	9.62	+5.35
/apollo/control	7.10	14.52	18.29	+3.78

Table 7: Accident Statistics. We classify accidents from three perspectives: ❶ exploitability, ❷ consequences, and ❸ driving scenarios. We specifically exclude accidents that are not exploitable or only result in minor consequences (e.g., inefficiency such as taking longer routes), categorizing the remaining cases.

Accident Sources	#Accident	❶ Exploitable	❷ Consequence			❸ Driving Scenario			
			Collision	Emergency Braking	Inefficiency	Intersection	Merging	Following	Turning
CROUTINE [72]	6	3 (50.0%)	1 (16.7%)	-	5 (83.3%)	-	-	3 (50.0%)	3 (50.0%)
CAT [71]	5	5 (100%)	2 (40.0%)	1 (20.0%)	2 (40.0%)	4 (80.0%)	-	1 (20.0%)	-
ATLAS [73]	10	7 (70.0%)	4 (40.0%)	2 (20.0%)	4 (40.0%)	9 (90.0%)	-	1 (10.0%)	-
DriveFuzz [52]	31	21 (67.7%)	18 (58.1%)	4 (12.9%)	9 (29.0%)	2 (6.5%)	4 (12.9%)	16 (51.6%)	9 (29.0%)
DoppelTest [42]	123	112 (91.1%)	98 (79.7%)	23 (18.7%)	2 (1.6%)	40 (32.5%)	81 (65.9%)	2 (1.6%)	-
PlanFuzz [77]	9	9 (100%)	-	6 (66.7%)	3 (33.3%)	3 (33.3%)	-	6 (66.7%)	-
Summary	184	157 (85.3%)	123 (66.8%)	36 (19.6%)	25 (13.6%)	58 (31.5%)	85 (46.2%)	29 (15.8%)	12 (6.5%)



(a) A1 (b) A3 (c) A4 (d) A5 (e) A6 (f) A10 (g) A11

Figure 14: Illustrations for accident scenarios in Table 2. Top images show simulator screenshots just before or at the accident moments. Blue vehicles are the ADSs. Solid lines show vehicles' trajectories. Vehicles without lines mean their speeds are 0.

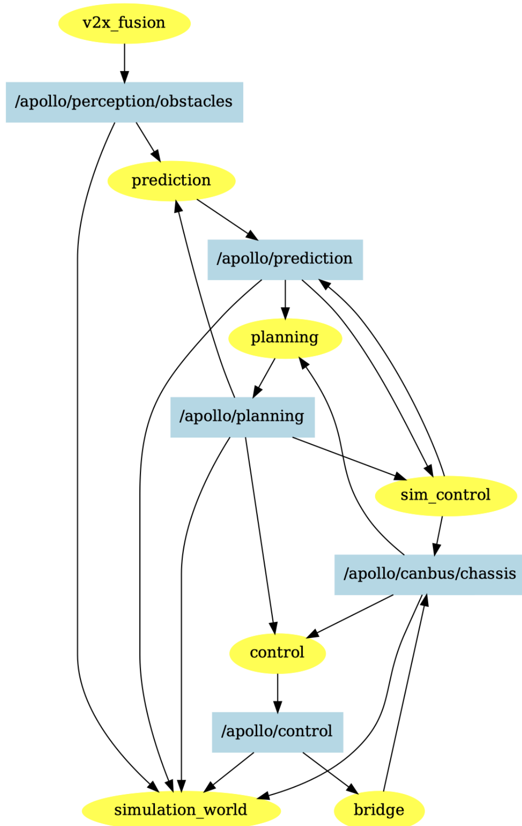


Figure 15: Extracted (simplified) CMG from CyberRT.

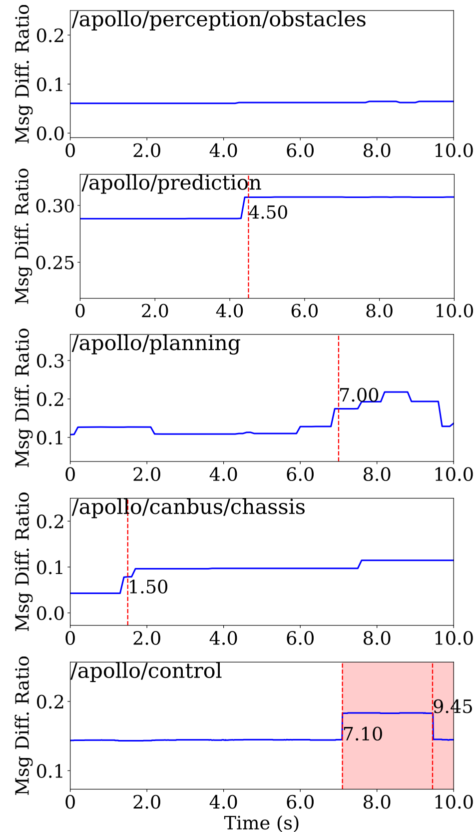


Figure 16: MDR series for CMG channels in Fig. 15. Red dot lines denote MDR change points. The accident happens at 7.1s, shown as red region in *control* channel subfigure.