A Comprehensive Study of Autonomous Vehicle Bugs

Joshua Garcia⋆, Yang Feng⋆,†, Junjie Shen⋆, Sumaya Almanee⋆, Yuan Xia⋆, and Qi Alfred Chen⋆

⋆University of California, Irvine, California, USA
†Nanjing University, Nanjing, China
{joshug4, yang.feng, junjies1, salmanee, yxia11, alfchen}@uci.edu

Abstract

Self-driving cars, or Autonomous Vehicles (AVs), are increasingly becoming an integral part of our daily life. About 50 corporations are actively working on AVs, including large companies such as Google, Ford, and Intel. Some AVs are already operating on public roads, with at least one unfortunate fatality recently on record. As a result, understanding bugs in AVs is critical for ensuring their security, safety, robustness, and correctness. While previous studies have focused on a variety of domains (e.g., numerical software; machine learning; and error-handling, concurrency, and performance bugs) to investigate bug characteristics, AVs have not been studied in a similar manner. Recently, two software systems for AVs, Baidu Apollo and Autoware, have emerged as frontrunners in the open-source community and have been used by large companies and governments (e.g., Lincoln, Volvo, Ford, Intel, Hitachi, LG, and the US Department of Transportation). From these two leading AV software systems, this paper describes our investigation of 16,851 commits and 499 AV bugs and introduces our classification of those bugs into 13 root causes, 20 bug symptoms, and 18 categories of software components those bugs often affect. We identify 16 major findings from our study and draw broader lessons from them to guide the research community towards future directions in software bug detection, localization, and repair.

Keywords

bugs, defects, autonomous vehicles, empirical software engineering

1 Introduction

Self-driving cars, or Autonomous Vehicles (AVs), are increasingly becoming an integral part of our daily life. For example, AVs are under rapid development recently, with some companies, e.g., Google

Waymo, already serving customers on public roads [6, 23, 24]. In total, there are already about 50 corporations actively developing AVs [1, 16]. AVs consist of both software and physical components working jointly to achieve driving automation in the physical world. Unfortunately, like any system relying upon software, they are susceptible to software bugs. As a result, faults or defects in such software are safety-critical, possibly leading to severe injuries to passengers or even death. For instance, an AV of Uber has already killed a pedestrian in 2018 [13, 19]. AVs with lower levels of autonomy have resulted in another set of fatalities during recent years [10, 12, 14, 15, 17, 18]. Given the safety-criticality of such vehicles, it is imperative that the software controlling AVs have minimal errors.

Unfortunately, the nature of AV software bugs is currently not well understood. It is unclear what the root causes of bugs are in AV software, the kinds of driving errors that may result, and the parts of AV software that are most often affected. These kinds of information can aid AV software researchers and engineers with (1) the creation of AV bug detection and testing tools, (2) the localization of faults that result in AV bugs, (3) recommendations or automated means of repairing AV bugs, (4) measurement of the quality of AV software, and (5) mechanisms to monitor for AV software failures.

Previous empirical studies have investigated bug characteristics in a variety of domains including numerical software libraries [32], machine learning libraries [38, 47, 55], concurrency bugs [40, 41], performance bugs [39, 46], and error-handling bugs [27, 29, 49]. None of these studies have focused on bugs in AV software systems.

This paper presents the first comprehensive study of bugs in AV software systems. Currently, there are two AV systems that achieve high levels of autonomy and have extensive issue repositories, i.e., Baidu Apollo [4] and Autoware [3]. Both of these systems have *representative* designs and are *practical*: For Baidu Apollo, its design is selected by Udacity to teach start-of-the-art AV technology [20], can be directly deployed on real-world AVs such as Lincoln MKZ [4], and has already reached mass production agreements with Volvo and Ford [5]. For Autoware, it is an open-source system for AVs run by the Autoware Foundation [3], whose members include a variety of industrial organizations, including Intel, Hitachi, LG, and Xilinx. Recently, Autoware has been selected by the USDOT (US Department of Transportation) to build their reference development platform for intelligent transportation solutions [11, 21].

We have studied 499 AV bugs from 16,851 commits across the Apollo and Autoware repositories. From a manual analysis of these bugs and commits, we have identified 13 root causes, 20 symptoms the bugs can exhibit, and 18 categories of AV software components that exhibit a significant amount of bugs. We further assess the relationships among the three phenomena. Based on these results,

we suggest future research directions for software testing, analysis, and repair of AV systems.

This paper makes the following contributions:

- We conduct the first comprehensive study of bugs in AV systems through a manual analysis of 499 AV bugs from 16,851 commits in the two dominant AV open-source software systems.
- We provide a classification of root causes and symptoms of bugs, and the AV components these bugs may affect.
- We discuss and suggest future directions of research related to software testing and analysis of AV systems.
- We make the resulting dataset from our study available for others to replicate or reproduce, or to allow other researchers and practitioners to build upon our work. Our artifacts can be found at the following website [9].

The rest of the paper is organized as follows. The next section provides further background on AV software systems. Section 3 discusses the methodology we use to conduct our study; the root causes, symptoms, and affected components we identified; and overviews and motivates the research questions we investigate. We then cover the results of our empirical study (Section 4), follow that with a discussion drawing broader lessons from those results (Section 5), and detail threats to validity (Section 6). Finally, we describe related work (Section 7) and conclude.

2 Autonomous Vehicle Systems

The Society of Automotive Engineers (SAE) defines 6 levels of vehicle autonomy [30], with *Level 0* (*L0*) being the lowest, i.e., no autonomy, and *Level 5* (*L5*) being the highest, i.e., full autonomy in any driving environment. *Level 4* (*L4*) is the highest autonomy level for which no human drivers are required to stay alert and ready to take over control anytime the system cannot make driving decisions. Compared to L5, L4's autonomy is limited to certain driving scenarios (e.g., certain geofenced areas), but it is already enough to enable a number of attractive use cases in practice such as highway driving, truck delivery, and fixed-route shuttles, while being easier to ensure safety than L5. Thus, nearly all AV companies aiming for high-level autonomy are focusing on L4 AV development, e.g., Google, Uber, Lyft, Baidu, GM Cruise, Ford, Aurora, TuSimple, etc. [1], and some of them are already available to the general public, e.g., the Google Waymo One self-driving taxi service [22]. In this work, we focus on L4 AV systems since they have the highest autonomy level among the AVs in production today and thus their software bugs and defects have the highest importance in terms of safety and robustness.

In L4 AV systems, software plays a central role to achieve intelligent driving functionality. For example, Fig. 1 shows the general AV software system architecture based on state-of-the-art designs referenced in most popular AV development classes such as Udacity Self-Driving Car Engineer classes [20] and used in representative real-world AV systems such as Baidu Apollo [4] and Autoware [3]. As shown, such a software system is in charge of *all the core decision-making steps* after receiving sensor input. Detailed functionality of each component in an AV software system is as follows:

- **Perception** processes LiDAR, camera, and radar inputs and detects obstacles such as vehicles and pedestrians. AV systems often adopt multiple object detection pipelines to avoid false detection. For example, Baidu Apollo consists of a camera-based and a LiDAR-based object-detection pipeline, which uses segmentation models based on Convolutional Neural Networks (CNNs). The detected obstacles from different pipelines are then fused together using algorithms such as a Kalman filter. Aside from detecting obstacles, the Perception component is also in charge of traffic light classification and lane detection.
- **Localization** provides an estimation of the AV's real-time location, which serves as the basis for driving decision-making. It accepts location measurements from GPS and LiDAR. Particularly, a LiDAR point cloud matching algorithm (e.g., NDT [26] and ICP [37]) finds the best match of the LiDAR input in a pre-built High-Definition Map (HD Map) to get the LiDAR-based location measurement. It then uses a multi-sensor fusion algorithm (e.g., Error State Kalman filter [52]) to fuse location measurements.
- **Prediction** estimates the future trajectory of the detected obstacles. Neural networks (e.g., MLPs and RNNs) are commonly used to evaluate the probabilities of the possible trajectories.
- **Planning** calculates the optimal driving trajectory considering factors such as safety, speed, and comfort. It incorporates various constraints (e.g., distances to obstacle trajectories, distance to lane center, smoothness of the trajectory, etc.) and solves a Linear Programming (LP) or Quadratic Programming (QP) problem to calculate the future trajectory that the AV needs to follow.
- **Control** enforces the planned trajectory with lateral and longitudinal control. It uses control algorithms such as MPC [35] and PID [25] to calculate the required steering and throttling.
- **CAN Bus** handles the underlying communication between the software and the vehicle to send control commands and receive chassis information.
- **Infrastructure** provides the necessary *utilities* and *tools* for the software, such as sensor calibration tools and CUDA [43]. It also includes a *robotics middleware* (e.g., ROS [44], Cyber RT [4]), which supports the communication among components.
- **High-Definition Map (HD Map)** is queried during runtime for information such as lane boundaries, traffic sign locations, stationary objects, routing, etc. Some AV systems, such as Baidu Apollo, use a centralized component called *Map Engine* to handle the queries; while others, such as Autoware, handle the map queries separately in each module.
- **Human Machine Interface (HMI)** collects and visualizes system status and interfaces with developers and passengers. This is not required for the autonomous driving function, but real-world AV software systems, e.g., those in both Apollo and Autoware, generally have it for usability.

3 Methodology and Classification

3.1 Data collection

We collect all commits, issues, pull requests of Apollo and Autoware that are created on or before July 15, 2019 via the GITHUB APIs as shown in Table 1. In total, we obtain 13,335 commits, 7,414 closed pull requests, and 9,216 issues for Apollo and collect 3,516 commits, 1,318 closed pull requests, and 2,314 issues for Autoware.

Given that the goal of this paper is to characterize defects of AV systems, we identify closed and merged pull requests that fix defects. Such pull requests allow us to (1) confirm that a bug or fix was
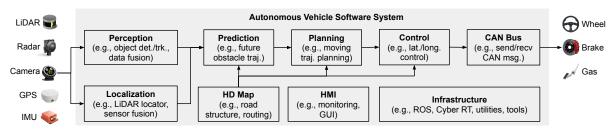
Figure 1: State-of-the-art Autonomous Vehicle (AV) software system architecture from most popular AV development classes such as Udacity Self-Driving Car Engineer classes [20] and real-world AV systems such as Baidu Apollo [4] and Autoware [3].

accepted by developers and (2) analyze the modified source code, related issues, and the discussion of developers. Note that, on GITHUB, pull requests are used for various purposes (e.g., new feature implementation, enhancement, and refactoring). To categorize the purpose of pull requests, developers often employ some keywords to tag them. However, because tagging is often project-specific, directly filtering bug-fix pull requests based on the tag may introduce bias. To avoid such bias, we employ a method that helps us to obtain as many bug-fix pull requests as possible. To that end, we adopt a method similar to that used in previous studies [32, 38, 50, 55] to identify bug-fix pull requests. Specifically, we set up a list of bug-related keywords, including *fix, defect, error, bug, issue, mistake, incorrect, fault, and flaw*, and then search for these words in both the tags and titles. If any tags or title of a pull request contain at least one keyword, we identify it as a bug-fix pull request. This process resulted in 336 and 430 merged pull requests for Apollo and Autoware that meet the criteria, respectively.

Table 1: Statistics of Apollo and Autoware from GitHub

System	Start Date : End Date	SLOC[1] (C/C++)	SLOC (Python)	Commits	Issues	Bugs
Apollo	07/04/2017–07/15/2019	323,624	20,956	13,335	9,216	243
Autoware	08/25/2015–06/13/2019	164,299	14,463	3,516	2,314	256

[1]SLOC: source lines of code

3.2 Classification and Labeling Process

To characterize AV defects, we focus on analyzing them from three perspectives: (1) the **root causes** that reflect the mistakes developers make in code; (2) the **symptoms** that bugs exhibit as represented by incorrect behaviors, failures, or errors during runtime; and (3) the AV **component** in which a bug resides.

Our manual analysis focused on merged pull requests because these types of issues contain the code changes, discussions, links to related issues, code reviews, and other information that can assist us with gaining a comprehensive understanding of bugs and their fixes.

To reduce the subjective bias during the labeling process, we assign each of the 336 and 430 merged bug-fix pull requests identified in the data collection step to two authors of this paper. Our process required each set of two authors to analyze the defect separately. They manually inspected the source code, commit messages, pull-request messages, and issue messages to identify the root causes, symptoms, and affected AV components.

Prior research has summarized the causes of software defects and bugs [45, 47, 51, 57]. In this paper, we initially adopted the taxonomy

of root causes presented in [45, 47] to analyze AV defects. We then enhanced that taxonomy by using an open-coding scheme to expand the list of root causes. Specifically, for pull requests whose root causes did not fit into the initial taxonomy, each author conducting the manual analysis selected her own label for the root cause. Once all the author's pull requests were labeled, she met with the author sharing her assigned pull requests to resolve differences in labeling. For bug symptoms, we followed a similar process, starting with an initial taxonomy of symptoms (e.g., crash and hang) derived from existing literature [32, 38, 47, 55]. We found multiple symptoms may arise per bug cause. A single issue may have multiple bug symptoms. Bugs may be counted twice if they fall under two different categories.

For AV components, labels were stable for top-level components (e.g., Planning and Localization). However, certain sub-components appeared frequently (e.g., object detection and multi-sensor fusion). As a result, for AV components, we also had authors meet to resolve discrepancies in labeling. Using this overall process resulted in a final list of 243 bugs in Apollo and 256 bugs in Autoware. Multiple issues may be mapped to the same pull request, root cause, symptom, and component. If we remove these duplicate issues, we have 211 bug instances for Autoware instead of 256.

3.3 Root Causes of AV Bugs

Using the process described in the previous section, the full list of root causes for AV bugs are as follows:

- **Incorrect algorithm implementation (Alg)**: The implementation of the algorithm's logic is incorrect and cannot be fixed by addressing only one of the other root causes.
- **Incorrect numerical computation (Num)**: This root cause involves incorrect numerical calculations, values, or usage.
- **Incorrect assignment (Assi)**: One or more variables is incorrectly assigned or initialized.
- **Missing condition checks (MCC)**: A necessary conditional statement is missing.
- **Data**: The data structure is incorrectly defined, pointers to a data structure are misused, or types are converted incorrectly.
- **Misuse of an external interface (Exter-API)**: This cause involves misuse of interfaces of other systems or libraries (e.g., deprecated methods, incorrect parameter settings, etc.)
- **Misuse of an internal interface (Inter-API)**: This cause involves misuse of interfaces of other components—such as mismatched calling sequences; violating the contract of inheritance; and incorrect opening, reading, and writing.
- **Incorrect condition logic (ICL)**: This occurs due to incorrect conditional expressions.

- **Concurrency (Conc)**: This cause involves misuse of concurrency-oriented structures (e.g., locks, critical regions, threads, etc.).
- **Memory (Mem)**: This cause involves misuse of memory (e.g., improper memory allocation or de-allocation).
- **Invalid Documentation (Doc)**: This cause involves incorrect manuals, tutorials, code comments, and text that is not executed by the AV system.
- **Incorrect configuration (Config)**: This cause involves modifications to files for compilation, build, compatibility, and installation (e.g., incorrect parameters in Docker configuration files).
- **Other (OT)** causes occur highly infrequently and do not fall into any one of the above categories.

3.4 Symptoms of AV Bugs

Using the process described earlier in this section, we obtained the following AV bug symptoms:
- **Crashes** terminate an AV system or component improperly.
- **Hangs** are characterized by an AV system or component becoming unable to respond to inputs while its process remains running.
- **Build** errors prevent correct compilation, building, or installation of an AV system or component.
- **Display and GUI (DGUI)** errors show erroneous output on a GUI, visualization, or the HMI of the AV system.
- **Camera (Cam)** errors prevent image capture by an AV camera.
- **Stop and parking (Stop)** errors refer to the incorrect behaviors occurring when the AV attempts to stop or park the vehicle (e.g., sudden stops at inappropriate times, failure to stop in emergency situations, and parking outside of the intended parking space).
- **Lane Positioning and Navigating (LPN)** errors involve incorrect behaviors shown in lane positioning and navigating (e.g., failing to merge properly into a lane and failing to stay in the same lane).
- **Speed and Velocity Control (SVC)** symptoms involve incorrect behaviors related to the control of vehicle speed and velocity (e.g., failure to enforce the planned velocity and failing to follow another vehicle at high speed).
- **Traffic Light Processing (TLP)** errors represent any incorrect behaviors involving handling of traffic lights.
- **Launch (Lau)** symptoms occur when an AV system or component fails to start.
- **Turning (Turn)** symptoms occur when an AV behaves incorrectly when making or attempting to make a turn (e.g., turning at the wrong angle and problems with turn signals).
- **Trajectory (Traj)** symptoms involve incorrect trajectory prediction results (e.g., incorrect trajectory angles or predicted paths).
- **IO** errors involve incorrect behaviors when performing inputs or outputs to files or devices.
- **Localization (LOC)** errors refer to incorrect behaviors related with multi-sensor fusion-based localization and may manifest as incorrect information on a vehicle's map.
- **Security & safety (SS)** symptoms involve behaviors affecting security or privacy properties (e.g., confidentiality, integrity, or availability), damage to the vehicle, or injury to its passengers.
- **Obstacle Processing (OP)** errors occur when AVs incorrectly process detected obstacles on the road (e.g., failure to correctly estimate distance from an object).

- **Logic** errors represent incorrect behaviors that do not terminate the program or fit into the aforementioned symptom categories.
- **Documentation (Doc)** symptoms include any errors in documentation including manuals, tutorial, code comments, and other text intended for human rather than machine consumption.
- **Unreported (UN)** symptoms cannot be identified by reading issue discussions or descriptions, source code, or issue labels.
- **Other (OT)** symptoms occur highly infrequently and do not fit into the above categories.

3.5 Affected AV Components

After the aforementioned labeling process, the following AV components (described in Section 2) had a significant amount of bugs: Perception, Localization, Prediction, HD Map, Planning, Control, and CAN Bus. Both systems structure directories into components shown in Figure 1 and share the same reference architecture. Table 2 shows other core components found to have a significant number of bugs after the labeling process was completed.

Table 2: Additional Core Components with Significant Bugs

Component	Description
Sensor Calibration	Checks, adjusts, or standardizes sensor measurements
Drivers	Contains the hardware drivers necessary for operating the AV
Robotics-MW	Contains robotics middleware
Utilities & Tools	Contains shared functionality that supports the core functionality of other components
Docker	Contains the Docker image housing an instance of the AV system
Documentation & Others	A catch-all component category for representing documentation and sub-components with secondary functionality that have few bugs and do not fit into other components.

Perception, Localization, and CAN Bus components had major sub-components with significant amounts of bugs. Table 3 depicts those sub-components.

Table 3: AV Sub-Components with Significant Bugs

Component	Sub-Component	Description
Perception	Object Detection	Identifies objects around the AV
	Object Tracking	Tracks object around the AV
	Data Fusion	Fuses data from different object-detection pipelines
Localization	Multi-Sensor Fusion	Fuses location measurements
	Lidar Locator	Obtains location measurements from Lidar
CAN Bus	Actuation	Handles CAN Bus operations involving vehicle actuation
	Communication	Handles general CAN Bus transmission and receipt of data
	Monitor	Tracks information exchanged across the CAN Bus

3.6 Research Questions

To conduct our study, we answer the following research questions that are concerned with root causes of AV bugs, the symptoms they exhibit, and the components affected by AV bugs.

Previous work that has extensively studied different types of bugs in other application domains have discussed different causes of bugs. Understanding such causes can aid in localizing a fault and is

necessary for creating correct fixes of bugs. Consequently, we study the following research question:

> **RQ1:** *To what extent do different root causes of AV bugs occur?*

The effects of the bugs themselves are critical for triaging them and assessing their impacts. In particular, the domain-specific symptoms of bugs, in this case as they involve AVs, are of special interest in this study. As a result, we study the following research question:

> **RQ2:** *To what extent do different AV bug symptoms occur?*

The kind and frequency of bug symptoms and their root causes are a first step toward better understanding bugs in AV systems. However, the extent to which a particular root cause may produce a specific symptom allows engineers to determine more actionable information as to how to address a bug. This leads us to study our next research question:

> **RQ3:** *What kinds of bug symptoms can each root cause produce?*

The reference architecture of an AV system allows us to better understand the manner in which functionality and processing is decomposed into components of a software system. Certain components may be more prone to bugs or are more important than others for bug identification and repair. This information further allows researchers to know which parts of an AV system require further effort in terms of predicting, detecting, and repairing bugs. Thus, we investigate the following research question:

> **RQ4:** *To what extent do AV components contain bugs?*

Closely related to **RQ4** are the specific symptoms that occur in AV components. Understanding the relationship between symptoms of bugs and the AV components they affect allows engineers to allocate more resources (e.g., developer time and effort) to the components that exhibit the most critical errors or contain the most risky faults. Due to Conway's law [31], i.e., the structure of a software system often reflects the groups of people working on the system, this relationship can also inform managers and technical leads as to how different bug symptoms will affect different teams of an AV system.

> **RQ5:** *To what extent do bug symptoms occur in AV components?*

4 Experimental Results

Given the previously described methodology, classification, and research questions, we now discuss our study's results.

4.1 RQ1: Root Causes

We begin discussing experimental results by covering the frequency of AV bugs' root causes in Apollo and Autoware, which is depicted in Table 4. For both AV systems, incorrect implementations of algorithms (Alg) and incorrect configurations (Config) are the most frequently occurring root causes: 74 bugs are due to incorrect algorithm implementations in Apollo and 65 in Autoware; 34 bugs are caused by incorrect configurations in Apollo and 102 in Autoware.

For incorrect algorithm implementations, repairing their resulting errors often requires non-trivial and extensive code modifications, potentially affecting many lines of code (i.e., 104 lines of code on average). As a result, localizing faults in these cases or automatically repairing them are likely to be highly challenging [36, 42, 53].

Table 4: Root Causes of Bugs in AV Systems

Root Cause	Apollo	Autoware	$Total_{cause}$
Algorithm (Alg)	74	65	139
Numerical (Num)	14	15	29
Assignment (Assi)	25	22	47
Missing Condition Checks (MCC)	16	4	20
Data	8	2	10
External Interface (Exter-API)	1	5	6
Internal Interface (Inter-API)	5	0	5
Incorrect Condition Logic (ICL)	17	13	30
Concurrency (Conc)	2	4	6
Memory (Mem)	6	9	15
Incorrect documentation (Doc)	36	13	49
Incorrect Configuration (Config)	34	102	136
Others	5	2	7
$Total_{system}$	**243**	**256**	**499**

> **Finding 1:** Incorrect algorithmic implementations, often involving many lines of code, cause 27.86% of AV bugs.

Incorrect configurations—which involve building, compilation, compatibility, and installation—receive a very high amount of attention in open-source AV systems. They are particularly frequent in Autoware with 102 such bugs occurring. This result indicates that configuring, compiling, ensuring compatibility, and enabling installations of AV systems is highly challenging and deserves greater attention by the software-engineering research community.

> **Finding 2:** Incorrect configurations causes a substantial number of AV bugs, i.e., 27.25% of such bugs.

Root causes of AV bugs that occur a relatively frequent amount but much less frequently than Alg, Config, or Doc causes are those involving improper assignments or initializations (Assi), incorrect condition logic (ICL), numerical issues (Num), and missing condition checks (MCC) with each category occurring a total of 47, 30, 29, and 20, respectively, across both systems. These kinds of root causes typically involve a relatively small number of lines of code (e.g., 20 lines or less) and are more amenable to existing fault localization and automatic program repair techniques [42].

> **Finding 3:** Root causes of bugs involving relatively few lines of code, i.e., 20 or fewer lines, cause 25.25% of bugs.

4.2 RQ2: AV Bug Symptoms

The next research question we discuss covers the symptoms that AV bugs exhibit. Table 5 shows the types of bug symptoms we identified that occur for Apollo and Autoware. Symptoms specific to the domain of AVs mainly involve errors related to driving, navigating, or localizing the vehicle itself or perceiving the environment around the vehicle. In total, these bug symptoms have 140 instances across both AV systems and specifically include the following types of symptoms: lane positioning and navigation; speed and velocity control; traffic-light processing; vehicle stopping, turning, trajectory, and localization; and obstacle processing. It is notable that Apollo's bugs exhibit significantly more of these types of symptoms. However, this does not necessarily indicate that Apollo has more driving bugs than

Autoware. Apollo developers may focus more on identifying and fixing these kinds of bugs than Autoware developers.

Table 5: Symptoms of Bugs in AV Systems

Symptom	Apollo	Autoware	$Total_{symp}$
Crash	24	29	53
Hang	1	2	3
Build	15	66	81
Camera (Cam)	2	7	9
Lane Positioning and Navigation (LPN)	20	5	25
Speed and Velocity Control (SVC)	26	16	42
Launch (Lau)	5	7	12
Traffic Light Processing (TLP)	6	1	7
Vehicle Stopping and Parking (Stop)	8	7	15
Vehicle Turning (Turn)	9	0	9
Vehicle Trajectory (Traj)	26	4	30
IO	2	8	10
Localization (Loc)	2	6	8
Obstacle Processing (OP)	3	1	4
Invalid Documentation (Doc)	36	13	49
Display and GUI (DGUI)	10	29	39
Security and Safety (SS)	3	2	5
Logic	33	24	57
Unreported (Un)	4	25	29
Others (OT)	8	4	12
$Total_{system}$	**243**	**256**	**499**

Among the driving bugs, the symptoms that occur most frequently involve speed and velocity control, trajectory, and lane positioning and navigation with 42, 30, and 25 total instances across both systems, respectively. These results indicate that these kinds of functionality are difficult to implement correctly. At the same time, they are also among the core functionality one would expect an AV to perform and are inherently safety-critical. For example, the following bug description was extracted from one of Autoware's issues where one developer noticed an unexpected behaviour of the steering control and velocity plan [1]:

> "Correction of angular velocity plan at the waypoint end...At the *WayPoint* end point, the steering angle control becomes unstable"

Software testing, bug detection and localization, and automatic repair for such AV bugs would likely be highly beneficial for the AV development community.

> **Finding 4:** 28.06% of bugs directly affect driving functionality of AVs with speed and velocity control, trajectory, and lane positioning and navigation occur the most frequently at 8.42%, 6.01%, and 5.01%, respectively.

Among the types of symptoms in our classification, the one that appears the most are actually build errors—with 15 in the case of Apollo and 66 in the case of Autoware. Autoware bugs resulting in build errors largely involve changes to upstream components. For instance, new versions of ROS are released requiring major changes to ensure compatibility in Autoware. Moreover, the build error differences might be related to the underlying build systems used by

[1] https://tinyurl.com/y5vd6mqu

Apollo and Autoware. In particular, Autoware uses the native ROS build system [44], which reuses the *CMake* syntax [8] when specifying the compilation configurations. Apollo, on the other hand, adopts the newer *Bazel* build system [7] developed by Google. However, given that both Autoware and Apollo are built on top of robotics middleware (e.g., ROS [44] or Cyber RT [4]), an interesting direction for future work includes determining what aspects of Autoware's design or the developers decision-making processes result in them making such extensive updates.

Bugs that crash an AV software system occur relatively frequently as well, with 53 occurrences across both AV systems. Note that the bug reports that specify these crashes rarely indicate whether or not they may directly affect the safe operation of the AV on a road. As a result, it is not clear that these crashes are necessarily safety-critical. For example, the reports do not identify whether the bugs would result in the vehicle stalling, being unable to move, stuck accelerating, etc. One interesting future research direction is determining the extent to which AV crash bugs result in safety or security-critical errors.

Logic errors occur frequently—with 57 instances in total for both AV systems. Often, there is no indication that there are any runtime errors that necessarily occur for the bugs reporting logic errors. However, developers, for this symptom, often report that there is enough potential for a runtime error occurring in the future.

Display or GUI errors are another frequent and notable type of symptom that occurs in both AV systems—totalling 39 instances. Apollo and Autoware each provide simulations or GUIs to allow the user or developer to configure or assess the functionality of an AV.

> **Finding 5:** Build errors, crashes, logic errors, and GUI errors are among the most frequently occurring domain-independent errors in AV systems amounting to 16.23% of bugs for build errors, 10.62% for crashes, 11.42% for logic errors, and 7.82% for GUI errors.

Along the lines of safety and security, our explicit category that denotes the number of bugs that are clearly identified as safety- or security-oriented only totals 5. We found very few instances where the bug reports clearly specify that a particular bug is in fact a definite safety or security issue in either AV system. In fact, many of the aforementioned driving bugs (e.g., speed and velocity control, trajectory, and lane positioning and navigation) are likely to be safety-critical. However, we conservatively marked bugs as security or safety issues only if the bug report clearly denotes the bug in question as being safety- or security-related. There is significant amount of work that should be conducted to further assess the safety and security properties of AV systems.

> **Finding 6:** Bugs reported with explicit safety or security symptoms occur highly infrequently, constituting only 1% of AV bugs.

4.3 RQ3: Causes and Symptoms

A better understanding of the relationship between root causes, symptoms, and the frequency at which a particular root cause may produce a specific symptom can guide engineers and researchers working on AVs to prevent, detect, localize, and fix AV bugs. To that end, we examine the results of RQ3.

Table 6 illustrates the extent to which a particular root cause resulted in a specific symptom across both AV systems. Recall that

Table 6: Frequency of symptoms that each root cause of a bug may exhibit across Apollo and Autoware.

Root Cause \ Symptom	Crash	Hang	Build	Cam	LPN	SVC	Lau	TLP	Stop	Turn	Traj	IO	Loc	OP	Doc	DGU	SS	Logic	Un	OT
Algorithm (Alg)	12	0	0	4	17	15	3	1	7	4	19	2	5	2	0	15	2	23	8	0
Numerical (Num)	1	0	0	0	2	4	0	0	0	3	4	0	1	0	0	3	0	9	2	0
Assignment (Assi)	5	0	1	2	1	6	0	1	2	2	4	3	1	0	0	4	0	11	2	2
Missing Condition Checks (MCC)	5	0	0	1	2	4	0	1	1	0	0	0	0	1	0	1	0	3	1	0
Data	1	0	1	0	0	0	0	3	0	0	1	0	1	0	0	0	0	1	0	2
External Interface (Exter-API)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	2	2
Internal Interface (Inter-API)	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	2	0	0	1	0
Incorrect Condition Logic (ICL)	4	0	0	0	3	8	0	1	3	0	1	1	0	1	0	1	0	4	3	0
Concurrency (Conc)	1	3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Memory (Mem)	10	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	1	1
Incorrect Documentation (Doc)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	49	0	0	0	0	0
Incorrect Configuration (Conf)	13	0	79	2	0	3	8	0	1	0	1	1	0	0	0	10	3	5	8	2
Others	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	2	0	0	1	3
Total	53	3	81	9	25	42	12	7	15	9	30	10	8	4	49	39	5	57	29	12

incorrect algorithm implementations were the most frequently occurring root cause in our classification scheme. Unsurprisingly, that cause resulted in a wide variety of symptoms, producing 16 out of 20 of the symptoms in our classification scheme. This root cause results in many symptoms directly affecting the correct driving of a vehicle (i.e., lane positioning and navigation, speed and velocity control, traffic-light processing, stopping and parking, vehicle turning and trajectory, localization, and obstacle processing). Symptoms especially affected by incorrect algorithm implementations include lane positioning and navigation (17 occurrences), speed and velocity control (15 occurrences), and trajectory (19 occurrences). This indicates that implementing such algorithms has a high complexity compared to other aspects of AV driving. Other symptoms that occur frequently due to incorrect algorithm implementations include crashes (12 occurrences), display and GUI errors (15 occurrences), and logic errors (23 occurrences). Given that many lines of code (i.e., 104 lines of code on average) often need to be added or modified to fix AV bugs arising due to incorrect algorithm implementations, a wide variety of AV-specific and safety-critical bugs are likely to be inapplicable for state-of-the-art fault localization and automatic program-repair techniques [36, 42, 53].

> **Finding 7:** Incorrect algorithm implementations involving many lines of code caused all 8 types of symptoms that directly affect the driving of a vehicle and caused 16 out of all 20 symptoms in our classification scheme.

The second-most frequently occurring cause is incorrect configurations involving compilation, building, compatibility, and installation (Config), as described in Section 4.1. Despite the total number of bugs due to this cause (136 instances) being similar to the number for incorrect algorithm implementations (139 instances), incorrect configurations only caused 13 out of 20 of the symptoms in our classification schema—with a vast majority of those symptoms being build errors, i.e., 79 out of 136 (58.09%). This result further reinforces that simply building or compiling such systems is highly non-trivial and can benefit from software-engineering research that aids in this process (e.g., bug detection and repair for handling upstream changes). Besides build errors, incorrect configurations caused a significant number of crashes, inability of components of

the AV system to launch (Lau), display and GUI errors (DGUI), and logic errors.

> **Finding 8:** Incorrect configurations caused a wide variety of bug symptoms, 13 out of 20, with a vast majority resulting in build errors, 79 out of 136 (58.09%)—indicating that properly configuring, building, and compiling these AV systems is a non-trivial maintenance effort.

Recall that bugs caused by incorrect assignments or initializations occurred relatively frequently across both AV systems, i.e., 47 instances out of a total of 499 (9.42%). This root cause produced 15 out of 20 of the symptoms in our classification scheme. This cause was particularly prominent for logic errors, crashes, display and GUI errors, IO errors, errors involving speed or velocity control, and trajectory errors. A relatively wide variety and significant amount of such errors may be automatically repaired or, at least, identified and localized using existing state-of-the-art approaches.

Misuse of conditional statements and incorrect condition logic mainly produced errors involving lane positioning and navigation, speed and velocity control, and crashes. Such root causes also resulted in logic errors that may lead to a future runtime error but did not necessarily occur at the time of the bug report. Fixing these combinations of bugs and symptoms often involve a relatively small number of changes to code, i.e., about 20 lines of code or less, making them particularly amenable to existing fault localization and automatic program-repair approaches.

> **Finding 9:** Incorrect assignments of variables, conditional statements, or condition logic caused 16 out of 20 AV bug symptoms.

Concurrency and memory errors are infrequently reported, indicating that they also likely occur infrequently in AV systems. Such root causes also had little effect on the actual successful driving of the vehicle, with only two instances occurring: A concurrency issue and a memory issue caused a bug involving speed and velocity control, respectively. However, no other driving symptom arose due to such potentially serious errors. Note that, as expected, memory errors did cause a reasonable number of crashes, i.e., 10 in our study.

> **Finding 10:** Concurrency and memory misuse caused relatively few bug symptoms, i.e., 21 out of 499 bugs (4.21%).

4.4 RQ4: Bug Occurrences in AV Components

In this section, we examine the frequency of bug occurrences in AV components of the reference architecture introduced in Section 2. Table 7 presents the number of occurrences for each top-level component, as described in Sections 2 and 3—or sub-component of the Perception, Localization, or CAN Bus components—for Apollo and Autoware. Bug occurrences for sub-components are shown if a significant number of bugs were found in them.

Table 7: Frequency of bug occurrences for each AV component

Component	Sub-Component	Apollo	Autoware	$Total_{comp}$
Perception	Object Detection	17	38	55
	Object Tracking	2	9	11
	Data Fusion	11	6	17
Localization	Multi-Sensor Fusion	9	21	30
	Lidar Locator	1	26	27
Trajectory Prediction		7	1	8
Map		13	5	18
Planning		93	42	135
Control		4	0	4
Sensor Calibration		11	11	22
Drivers		3	15	18
CAN Bus	Actuation	4	2	6
	Communication	2	0	2
	Monitor	4	2	6
Robotics-MW		1	6	7
Utilities and Tools		12	41	53
Docker		7	6	13
Documentation and Others		42	25	67
$Total_{system}$		**243**	**256**	**499**

The number of bugs found in the Planning components of the AV systems far exceed that of others, totalling 135 bugs out of 499 (i.e., 27.05% of all bugs). For comparison, the second-most bug-ridden component type, Perception, only contains 83 bugs across both systems (i.e., 16.63% of all AV bugs)—resulting in Planning having 61.48% more bugs than Perception. It is reasonable that developers focus a significant amount of their effort on Planning because it makes major driving decisions about the safety, speed, and passenger comfort of an AV.

Bugs are generally found in three Perception sub-components: object detection, object tracking, and data fusion. The number of bugs in object detection (55) far exceeds the number of bugs found in either object tracking (11) or data fusion (17). Perception must handle sensor input from a variety of sources (e.g., LiDAR, camera, and radar) and use complex algorithms (e.g., Convolutional Neural Networks and Kalman filters). The module must also detect obstacles, classify traffic lights, and detect lanes. Due to this complexity, it is sensible for Perception to have such a high number of bugs.

Following Perception in terms of bug occurrences are Localization components, which account for 57 bugs out of 499 total (11.42%). Localization estimates an AV's real-time location based on a variety of location measurements and fuses them together. Multi-sensor fusion and lidar locator sub-components each have a similar number of bugs across both systems with 30 instances and 27 instances, respectively.

> **Finding 11:** The core AV components with the greatest number of bugs across both systems are Planning, Perception, and Localization—ordered from most bug-ridden to least—with 135 (27.05%), 83 (16.63%), and 57 (11.42%) bugs, respectively.

A substantial number of bugs involve functionality that does not provide core logic that fits into the major components as described in Section 2. 53 out of 499 bugs (10.62%) occur in components that provide utilities or tools that support core functionality and are often used by a variety of other components. For example, Figure 2 was obtained from one of Autoware's issues[2] and it shows a bug related to the runtime manager, which is one of the utilities responsible for starting and terminating Autoware's functional components. This bug prevented the runtime manager parameters from getting saved. Another significant number of bugs are either documentation-oriented, or occur infrequently and do not fit into other component categories, constituting 67 bugs across both AV systems (13.43%).

> **Finding 12:** Many bugs do not occur in the core domain-specific functionality of AV systems—constituting 53 bugs (10.62%) in the case of utilities and 67 bugs (13.43%) involving documentation bugs or bugs that do not occur frequently enough to fall into a major component category.

Figure 2: A bug found in one of Autoware's utilities.

4.5 RQ5: Bug Symptoms in AV Components

The final research question we study relates symptoms of bugs with the AV components they affect. Studying such a relationship allows engineers to better distribute engineering effort and other development resources (e.g., testing budget) to the components that exhibit the most bugs or the bug types that are of greatest importance to AV stakeholders.

Table 8 illustrates the extent to which different bug symptoms occur in components and sub-components across both Apollo and Autoware. Symptoms involving driving and operation of the vehicle are largely associated with bugs in Planning. Specifically, out of 140 bugs directly affecting driving of the vehicle, 90 of them affect Planning. Five particular driving symptoms—i.e., lane positioning and navigation (LPN), speed and velocity control (SVC), stopping and parking (Stop), vehicle turning (Turn) and trajectory (Traj)—that appear particularly frequently in Planning constitute 87 of the 140 driving bugs (62.14%). As an example, the following code snippet from Apollo illustrates a driving bug in Planning [3]:

```
if (init_trajectory_point_.v() <
```

---

[2] https://tinyurl.com/yxskk46n
[3] https://tinyurl.com/y4v4l7mc

Table 8: Occurrences of bug symptoms in components of Apollo and Autoware

Component	Sub-Component	Crash	Hang	Build	Cam	LPN	SVC	Lau	TFP	Stop	Turn	Traj	IO	Loc	OP	Doc	DGUI	SS	Logic	UN	OT
Perception	Object Detection	12	0	16	1	1	3	2	4	0	0	0	1	0	0	0	5	0	4	6	0
	Object Tracking	3	1	2	0	2	0	0	1	0	0	0	0	0	0	0	0	0	2	0	0
	Data Fusion	5	0	1	1	0	0	1	0	0	0	0	0	0	1	1	1	0	2	4	0
Localization	Multi-Sensor Fusion	3	1	7	4	0	2	1	0	0	0	0	0	5	0	0	1	0	3	3	0
	Lidar Locator	8	0	5	0	0	2	0	0	0	0	1	0	3	0	2	0	0	4	1	1
	Prediction	0	0	0	0	1	1	0	0	0	0	5	0	0	0	0	0	0	1	0	0
	Map	0	0	1	0	4	1	1	1	0	0	3	0	0	0	1	0	0	5	1	0
	Planning	8	0	4	0	17	29	0	1	14	6	21	3	0	2	4	4	1	12	8	1
	Control	0	0	0	0	0	2	0	0	0	1	0	0	0	0	0	0	0	1	0	0
	Calibration	2	0	1	1	0	0	1	0	0	0	0	2	0	1	3	3	0	6	0	2
	Drivers	0	0	4	1	0	0	0	0	0	0	0	3	0	0	0	4	0	3	1	2
CAN Bus	Actuation	0	0	2	0	0	1	0	0	0	2	0	0	0	0	0	0	0	0	0	1
	Communication	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
	Monitor	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
	Robotics-MW	2	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0
	Utilities	6	0	12	1	0	0	3	0	0	0	0	1	0	0	0	16	2	9	2	1
	Docker	0	0	7	0	0	0	1	0	0	0	0	0	0	0	0	0	1	2	1	1
Documentation & Others		2	0	16	0	0	1	1	0	1	0	0	0	0	0	38	4	0	1	1	2

```
qp_spline_path_config_.uturn_speed_limit() &&
  !is_change_lane_path_ &&
qp_spline_path_config_.reference_line_weight() > 0.0)
```

Specifically, the first conditional statement ensures that the current speed is less than the speed limit enforced for a U-turn.

Besides the sheer number of bugs in Planning identified in the previous section, the symptoms of bugs exhibited in the component indicate its high importance for assuring an AV system with low errors and high quality.

> **Finding 13:** Planning components have both a high number of bugs and exhibit many symptoms that are particularly important for safe and correct driving of AVs (62.14% of driving bugs).

Many bugs that crash AV components occur mainly in Perception (20 bugs), Localization (11 bugs), Planning (8 bugs), and Utilities (6 bugs). The fact that AVs can potentially be crashed substantially through the component that processes sensor inputs (i.e., Perception) is particularly concerning: A skilled malicious adversary with enough time may be able to turn a crash into an attack. Additionally, crashing components that let the AV know its position in the environment (i.e., Localization) or even prevent it from making decisions (i.e., Planning) may cause the AV to stop functioning, think it is somewhere it is not, or make dangerous decisions.

> **Finding 14:** Crash bugs occur throughout critical AV components—especially Perception, Localization, and Planning—making them susceptible to more dangerous secondary effects.

Build errors affect the overwhelming majority of AV component types, i.e., 16 out of 18 in our classification scheme. This result further corroborates the non-trivial nature of properly building and compiling AV systems, providing further evidence that solving and automating this challenge is an open and important research problem.

> **Finding 15:** Build errors affect many components, 15 out of 18 (83.33%).

Few bug symptoms affect 8 more components or sub-components, i.e., more than 40% of components in our classification scheme. Besides crashes and build errors, the only symptoms that occur that frequently include speed and velocity control (SVC), display and GUI errors (DGUI), and logic errors. SVC bugs, in particular, span 9 or more components, which is quite high for a domain-specific symptom focused on a particular type of functionality of AVs. DGUI and logic errors are relatively general and not domain-specific, so their high occurrence across components is less surprising.

> **Finding 16:** Bugs exhibiting speed and velocity control errors affect a significant number of AV components, i.e., 9 out of 18 components (50.00%) .

5 Discussion

Using the major findings of the previous section, we will discuss the larger implications of our study's results. In particular, we will draw lessons from the findings that can guide future work in areas related to software testing, analysis, and repair of AVs.

Findings 2. 5, 8, and 15 involve incorrect configurations or build errors. To summarize those findings, 27.25% of bugs occur due to incorrect configurations, while 16.23% of bugs result in build errors. Incorrect configurations cause a wide variety of symptoms (13 out of 20) and build errors affect many components (15 out of 18). This suggests that a major amount of time and effort expended by engineers are spent simply dealing with compatibility and compilation issues, upstream changes, and ensuring proper installation. If software-engineering research can aid engineers with such a problem, this would potentially open up a substantial amount of time

that engineers can spend actually ensuring safe, secure, and correct autonomous-driving functionality.

Incorrect algorithm implementations cause many bugs (27.86% of AV bugs), often involve many lines of code (104 lines of code on average), and cause many symptoms (16 out of 20), including all eight domain-specific driving symptoms, as corroborated by Findings 1 and 7. These findings strongly suggest that existing bug localization and repair approaches likely need to be augmented with domain-specific information and the ability to conduct repairs that involve many lines of non-trivial code.

Despite this, there is evidence that a significant amount of bugs may be applicable to existing bug detection, localization, and repair techniques. Specifically, bugs involving relatively few lines of code constitute about 25.25% of bugs (Finding 3) and simpler bugs (e.g., incorrect assignments and condition logic, and missing condition checks) cause 16 out of 20 AV bug symptoms (Finding 9). These results indicate that our study's dataset, consisting of AV bug causes and symptoms, would significantly aid researchers with assessing or constructing bug detection, localization, and repair techniques.

With few bugs being reported that explicitly identify safety and security concerns in AV systems (Findings 6), it is unclear how safe or secure open-source AV systems are. This strongly indicates the need for researchers to focus more effort on assessing and ensuring these critical properties in AV software. Particularly, certain bugs (e.g., crashes in Finding 14) may have security and safety implications and should be further explored in future work.

Software testing and analysis research for AVs have heavily focused on only a sub-component of Perception, i.e., object detection [48, 54]. However, our study provides significant evidence that many bugs, especially those that most involve actually driving the vehicle, occur in many other components (Findings 15 and 16)—especially Planning and Localization (Findings 11 and 13). With Planning having the overwhelmingly largest proportion of bugs (62.14%) and affecting 7 out of 8 driving symptoms, it is arguable that researchers should focus more on Planning than Perception. Even in the case of Perception, object tracking and data fusion, two sub-components that are not studied in terms of software testing and analysis, as far as we are aware, constitute 33.73% of bugs in Perception—while object detection covers the remaining Perception bugs.

6 Threats to validity

Internal threats. The primary internal threat to validity involves subjective bias or errors in classification of bugs. To reduce this threat, we initiate our labelling process with classification schemes from existing literature [45, 47], and adopt an open-coding scheme to assist us in expanding the initial schemes. To ensure that we focus on real bugs and fixes, we selected only accepted and closed issues (e.g., when determining issues related to a pull request), or merged and closed pull requests, and used bug tags when available. Further, each bug is inspected and labelled by two authors independently. Any discrepancy is discussed until a consensus is reached.

When issues, pull requests, or code were insufficient for facilitating classification into a particular category, we assigned the corresponding bug to the OT (Other) category. We also had discussions with Apollo developers about their bugs, allowing us to improve our classification, reducing subjective bias and error in the process.

External threats. One external threat is the generalizability of the data set we collected. We have adopted several strategies to mitigate this threat. First, the raw data we collected includes all pull requests, commits, and issues from the creation of subject AV systems until July 15, 2019. This strategy assures this study covers a comprehensive set of data. Second, we have adopted a method similar to those used in existing bug studies [32, 38, 50, 55] to identify as many bug-fix pull requests as possible in the data pre-processing step. Third, we have only studied the merged pull requests that fix bugs to ensure that the studied bugs, as well as their corresponding fixes, are accepted by developers.

Another threat to external validity is the generalizability of our findings. We study two AV systems, Apollo and Autoware, which are developed by two independent groups. Although there are only two systems in our study, they are the most widely used open-source AV systems containing over 520,000 lines of code, over 16,000 commits, and more than 10,000 issues. Additionally, these AV systems are used by about 50 corporations and governments—including the US government, Google, Intel, Volvo, Ford, Hitachi, and LG [1, 5, 6, 11, 16, 21, 23, 24]. Furthermore, the number of labeled bugs (499 in this study) is similar in size to that of other recent bug studies in other domains (e.g., 555 [38] and 175 [55] for deep learning and 269 for numerical libraries [32]). Given that we focus on L4 AV systems, Apollo and Autoware are the only systems that achieve that high level of autonomy and have extensive, as well as publicly accessible, issue repositories [2, 3]. Given the high level of autonomy and sizes of data in our study, its findings are more likely to be representative and generalizable to other AV software aiming for L4 autonomy.

7 Related Work

Empirical study on bugs. A great number of work has been conducted that studies bugs in different types of software systems. Franco *et al.* [32] studied the bugs that occur in numerical software libraries such as NumPy, SciPy, and LAPACK. Islam *et al.* [38], Thung *et al.* [47], and Zhang *et al.* [55] investigated machine learning and deep learning frameworks (e.g., Caffe, Tensorflow, OpenNLP, etc.). Leesatapornwongsa *et al.* [40] and Lu *et al.* [41] analyzed concurrency bugs in distributed systems such as Hadoop and HBase. Jin *et al.* [39] and Selakovic *et al.* [46] studied performance bugs in large-scale software suites such as Apache, Chrome, and GCC. Different from this prior work, we are the first to perform an empirical study on bugs in emerging AV software systems.

Across existing empirical studies [32, 38–41, 46, 47, 55], bugs are often characterized based on multiple dimensions including bug types, root causes, and bug symptoms. Some work has categorized bugs using domain-specific characteristics, such as the triggering of timing conditions for concurrency bugs [41]. Compared to this prior work, we apply and adapt these bug characterization methods to bugs in a different domain, i.e., AV software systems.

AV software robustness. For AVs, ensuring the robustness of its software system is the top priority—as any software bugs may incur serious damage to both the road entities and the AV itself. Unfortunately, many fatal accidents have occurred in recent years due to the lack of software robustness. For example, Tesla's autopilot system has been the culprit of several deaths over recent years [10, 12, 14, 15, 17, 18]. Uber's AV system reportedly failed to prevent

the crash after detecting the pedestrian 6 seconds before the accident in Tempe, AZ [13, 19]. Moreover, machine learning models used in AV systems (e.g., in Perception) have been found vulnerable to attacks (e.g., physical-world perturbations [33, 34, 56] or sensor attacks [28]). Compared to these case-by-case discoveries of AV-system robustness issues, we are the first to systematically collect, taxonomize, and characterize bugs in AV systems, which is a critical first step towards eliminating them in a principled manner.

8 Conclusion

AV systems are increasingly operating throughout our daily lives. Given the fact that AV systems are designed for safety-critical tasks, it is vital that software-engineering researchers build robust quality-assurance techniques and methods for them. In this paper, we characterized and taxonomized bugs for AV systems—identifying 13 root causes and 20 symptoms across 18 components for 499 bugs from Apollo and Autoware, the only two open-source AV systems achieving high levels of autonomy (L4). Both researchers and developers of the AV systems can benefit from this study. For developers, we summarize 16 findings from this study to help them deal with bugs. For researchers, we distill challenges from our findings that call for more research effort. Our findings offer both developers and researchers a principled and improved understanding of AV bugs.

For future work, we aim to reproduce bugs we found in this study. Achieving this reproduction requires overcoming numerous research challenges including understanding and specifying driving scenarios in a simulator, and controlling it carefully to precisely trigger bugs. Additionally, we aim to construct techniques for automatic test-case generation and test-oracle construction for AV systems and assessing the degree to which fuzzing, search-based testing, and symbolic execution are applicable to AV software systems. Lastly, we are refining our study's results to produce a benchmark dataset for facilitating automatic program repair for AV software systems.

9 Acknowledgement

References

[1] August 2019. 46 Corporations Working On Autonomous Vehicles. https://www.cbinsights.com/research/autonomous-driverless-vehicles-corporations-list/.

[2] August 2019. Apollo Cyber RT. https://github.com/ApolloAuto/apollo/tree/master/cyber.

[3] August 2019. Autoware: Open-source software for urban autonomous driving. https://github.com/CPFL/Autoware.

[4] August 2019. Baidu Apollo: An open autonomous driving platform. http://apollo.auto/.

[5] August 2019. Baidu hits the gas on autonomous vehicles with Volvo and Ford deals. https://techcrunch.com/2018/11/01/baidu-volvo-ford-autonomous-driving/.

[6] August 2019. Baidu starts mass production of autonomous buses. https://www.dw.com/en/baidu-starts-mass-production-of-autonomous-buses/a-44525629.

[7] August 2019. Bazel. https://bazel.build/.

[8] August 2019. CMake. https://cmake.org/.

[9] August 2019. A Comprehensive Study of Autonomous Vehicle Bugs - Artifact Website. http://tiny.cc/cps_bug_analysis.

[10] August 2019. Fatal Tesla Crash Exposes Gap In Automaker's Use Of Car Data. https://www.forbes.com/sites/alanohnsman/2018/04/16/tesla-autopilot-fatal-crash-waze-hazard-alerts/#7bb735fb5572.

[11] August 2019. Github: The CARMA platform. https://github.com/usdot-fhwa-stol/CARMAPlatform.

[12] August 2019. Self-Driving Tesla Was Involved in Fatal Crash, U.S. Says. https://www.nytimes.com/2016/07/01/business/self-driving-tesla-fatal-crash-investigation.html.

[13] August 2019. Self-Driving Uber Car Kills Pedestrian in Arizona, Where Robots Roam. https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html.

[14] August 2019. Tesla: Autopilot was on during deadly Mountain View crash. https://www.mercurynews.com/2018/03/30/tesla-autopilot-was-on-during-deadly-mountain-view-crash/.

[15] August 2019. Tesla driver dies in first fatal crash while using autopilot mode. https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk.

[16] August 2019. The 18 Companies Most Likely to Get Self-driving Dars on the Road First. https://www.businessinsider.com/the-companies-most-likely-to-get-driverless-cars-on-the-road-first-2017-4.

[17] August 2019. There are some scary similarities between Tesla's deadly crashes linked to Autopilot . https://qz.com/783009/the-scary-similarities-between-teslas-tsla-deadly-autopilot-crashes/.

[18] August 2019. Two Years On, A Father Is Still Fighting Tesla Over Autopilot And His Son's Fatal Crash . https://jalopnik.com/two-years-on-a-father-is-still-fighting-tesla-over-aut-1823189786.

[19] August 2019. Uber Self-Driving Car Crash: What Really Happened. https://www.forbes.com/sites/meriameberboucha/2018/05/28/uber-self-driving-car-crash-what-really-happened.

[20] August 2019. Udacity: Self-Driving Fundamentals: Featuring Apollo. https://www.udacity.com/course/self-driving-car-fundamentals-featuring-apollo--ud0419.

[21] August 2019. USDOT: The CARMA platform. https://highways.dot.gov/research/research-programs/operations/CARMA.

[22] August 2019. Waymo Launches Self-driving car Service Waymo One. https://techcrunch.com/2018/12/05/waymo-launches-self-driving-car-service-waymo-one.

[23] August 2019. Waymo's autonomous cars have driven 8 million miles on public roads. https://www.theverge.com/2018/7/20/17595968/waymo-self-driving-cars-8-million-miles-testing.

[24] August 2019. You can take a ride in a self-driving Lyft during CES. https://www.theverge.com/2018/1/2/16841090/lyft-aptiv-self-driving-car-ces-2018.

[25] Karl J Åström and Björn Wittenmark. 2013. Adaptive control. Courier Corporation.

[26] Peter Biber and Wolfgang Straßer. 2003. The normal distributions transform: A new approach to laser scan matching. In Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453), Vol. 3. IEEE, 2743–2748.

[27] N. Cacho, E. A. Barbosa, J. Araujo, F. Pranto, A. Garcia, T. Cesar, E. Soares, A. Cassio, T. Filipe, and I. Garcia. 2014. How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications. In 2014 IEEE International Conference on Software Maintenance and Evolution. 31–40. https://doi.org/10.1109/ICSME.2014.25

[28] Yulong Cao, Chaowei Xiao, Dawei Yang, Jing Fang, Ruigang Yang, Mingyan Liu, and Bo Li. 2019. Adversarial Objects Against LiDAR-Based Autonomous Driving Systems. arXiv preprint arXiv:1907.05418 (2019).

[29] R. Coelho, L. Almeida, G. Gousios, and A. v Deursen. 2015. Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues. In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. 134–145. https://doi.org/10.1109/MSR.2015.20

[30] SAE On-Road Automated Vehicle Standards Committee et al. 2014. Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems. SAE Standard J 3016 (2014), 1–16.

[31] Melvin E Conway. 1968. How do committees invent. Datamation 14, 4 (1968), 28–31.

[32] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A Comprehensive Study of Real-world Numerical Bug Characteristics. In Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, Piscataway, NJ, USA, 509–519. http://dl.acm.org/citation.cfm?id=3155562.3155627 event-place: Urbana-Champaign, IL, USA.

[33] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramer, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. Physical Adversarial Examples for Object Detectors. In USENIX Workshop on Offensive Technologies (WOOT).

[34] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2017. Robust physical-world attacks on deep learning models. arXiv preprint arXiv:1707.08945 (2017).

[35] Carlos E Garcia, David M Prett, and Manfred Morari. 1989. Model predictive control: theory and practiceâĂȚa survey. Automatica 25, 3 (1989), 335–348.

[36] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 19–30.

[37] Dirk Holz, Alexandru E Ichim, Federico Tombari, Radu B Rusu, and Sven Behnke. 2015. Registration with the point cloud library: A modular framework for aligning in 3-D. *IEEE Robotics & Automation Magazine* 22, 4 (2015), 110–124.

[38] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. *arXiv:1906.01388 [cs]* (June 2019). http://arxiv.org/abs/1906.01388 arXiv: 1906.01388.

[39] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075 event-place: Beijing, China.

[40] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374 event-place: Atlanta, Georgia, USA.

[41] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323 event-place: Seattle, WA, USA.

[42] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering* 23, 5 (01 Oct 2018), 2901–2947. https://doi.org/10.1007/s10664-017-9550-0

[43] Nvidia. 2010. CUDA Programming guide.

[44] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[45] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. 2008. Defect Categorization: Making Use of a Decade of Widely Varying Historical Data. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08)*. ACM, New York, NY, USA, 149–157. https://doi.org/10.1145/1414004.1414030

[46] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2884781.2884829 event-place: Austin, Texas.

[47] F. Thung, S. Wang, D. Lo, and L. Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 271–280. https://doi.org/10.1109/ISSRE.2012.22

[48] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *International Conference on Software Engineering (ICSE)*.

[49] Yuchi Tian and Baishakhi Ray. 2017. Automatically Diagnosing and Repairing Error Handling Bugs in C. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 752–762. https://doi.org/10.1145/3106237.3106300 event-place: Paderborn, Germany.

[50] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 805–816.

[51] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug characteristics in blockchain systems: a large-scale empirical study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 413–424.

[52] Greg Welch and Gary Bishop. 1995. An introduction to the Kalman Filter. (1995).

[53] X. Xia, L. Bao, D. Lo, and S. Li. 2016. âĂIJAutomated Debugging Considered HarmfulâĂİ Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 267–278. https://doi.org/10.1109/ICSME.2016.67

[54] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/3238147.3238187

[55] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866 event-place: Amsterdam, Netherlands.

[56] Yue Zhao, Hong Zhu, Ruigang Liang, Qintao Shen, Shengzhi Zhang, and Kai Chen. 2018. Seeing isn't Believing: Practical Adversarial Attack Against Object Detectors. *arXiv preprint arXiv:1812.10217* (2018).

[57] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. 2015. An empirical study on quality issues of production big data platform. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 17–26.