

UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

Assuring Software Dependability of Smart Systems

Permalink

<https://escholarship.org/uc/item/1ss0464x>

Author

Almanee, Sumaya

Publication Date

2022

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at <https://creativecommons.org/licenses/by/4.0/>

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

Assuring Software Dependability of Smart Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Sumaya Almanee

Dissertation Committee:
Assistant Professor Joshua Garcia, Chair
Professor Sam Malek
Assistant Professor Qi Alfred Chen

2022

DEDICATION

To Hadeel,

صديقة العمر ورفيقة الدرب

For always being by my side ... Here's to another 10 years of true friendship ...

أحبك هدولة قلبي

To my kitty cat ... Java,

For staying up late when I was working on my deadlines, for poking your tiny head during my zoom meetings and for comforting me with your purrs and cuddles ...

I love you kiki ...

To family, friends and loved ones ...

TABLE OF CONTENTS

	Page
LIST OF FIGURES	vi
LIST OF TABLES	vii
ACKNOWLEDGMENTS	ix
VITA	xi
ABSTRACT OF THE DISSERTATION	xiii
1 Introduction	1
1.1 Security Updates in Android Apps' Native Code	3
1.2 Analysis of Bugs in Autonomous Vehicles	5
1.3 Test Generation for Autonomous Vehicles	5
1.4 Dissertation Structure	6
2 Research Problem	9
2.1 Problem Statement	9
2.2 Research Hypothesis	10
2.2.1 Software Security in Smart Systems	10
2.2.2 Software Reliability in Smart Systems	11
2.2.3 Software Safety in Smart Systems	11
3 An Empirical Study of Security Updates in Android Apps' Native Code	13
3.1 Introduction	14
3.2 <i>LibRARIAN</i>	17
3.2.1 Feature Vector Extraction	18
3.2.2 Similarity Computation	20
3.2.3 Version Identification Strings	22
3.3 Evaluation	23
3.3.1 RQ1: Accuracy and Effectiveness	24
3.3.2 RQ2: Prevalence of Vulnerable Libraries	29
3.3.3 RQ3: Rate of Vulnerable Library Fixing	32
3.3.4 Exploitability Case Study	37
3.4 Discussion	38

3.5	Threats to Validity	39
3.6	Related Work	41
4	A Comprehensive Study of Autonomous Vehicle Bugs	44
4.1	Introduction	45
4.2	Autonomous Vehicle Systems	47
4.3	Methodology and Classification	50
4.3.1	Data collection	50
4.3.2	Classification and Labeling Process	51
4.3.3	Root Causes of AV Bugs	52
4.3.4	Symptoms of AV Bugs	53
4.3.5	Affected AV Components	55
4.3.6	Research Questions	56
4.4	Experimental Results	57
4.4.1	RQ1: Root Causes	57
4.4.2	RQ2: AV Bug Symptoms	59
4.4.3	RQ3: Causes and Symptoms	62
4.4.4	RQ4: Bug Occurrences in AV Components	65
4.4.5	RQ5: Bug Symptoms in AV Components	67
4.5	Discussion	70
4.6	Threats to validity	71
4.7	Related Work	73
5	Generating Diverse, Fully-Mutable, Safety-Critical and Motion Sickness-Inducing Scenarios for Autonomous Vehicles	75
5.1	Introduction	76
5.2	Related Work	81
5.3	Specification of the State Space	86
5.4	SCENORITA	88
5.4.1	Domain-Specific Constraints	88
5.4.2	Scenario Generator	91
5.4.3	Generated Scenarios Player	100
5.4.4	Planning Output Recorder	101
5.4.5	Grading Metrics Checker	101
5.4.6	Duplicate Violations Detector	104
5.5	Evaluation	107
5.5.1	Experiment Settings	107
5.5.2	RQ1: Accuracy of Generated Scenarios	111
5.5.3	RQ2: Effectiveness at Producing Scenarios with Safety and Comfort Violations	113
5.5.4	RQ3: Efficiency of SCENORITA	119
5.5.5	RQ4: Duplicate Violation Detection	121
5.6	Threats to Validity	123
5.7	Discussion	125

6 Conclusion	126
6.1 Research Contributions:	127
6.2 Future Work:	129
Bibliography	132

LIST OF FIGURES

	Page
3.1 <i>LibRARIAN</i> identifies versions of native binaries from Android apps by using our <i>bin²sim</i> similarity-scoring technique to compare known (ground-truth dataset) and unknown versions of native binaries.	17
4.1 State-of-the-art Autonomous Vehicle (AV) software system architecture from most popular AV development classes such as Udacity Self-Driving Car Engineer classes [34] and real-world AV systems such as Baidu Apollo [15] and Autoware [13].	47
4.2 A bug found in one of Autoware’s utilities.	67
5.1 An Overview of SCENORITA	90
5.2 Genetic representation of: (a) fully-mutable tests in SCENORITA, and (b) partially mutable tests generated by state-of-the-art approaches.	92
5.3 (a) two individuals <i>before</i> a crossover, (b) the same individuals <i>after</i> a crossover for SCENORITA, and (c) how crossover is applied in prior work [130]	98
5.4 An example of a crossover that produced individuals with invalid attributes (highlighted in red)	99
5.5 (a) mutating a gene in a <i>single</i> individual, (b) mutating a scenario by adding a fit individual from another scenario, and (c) mutating a scenario by removing the worst individual.	100
5.6 An illustration of one of five supported driving scenarios in AutoFuzz. (a) The ego car (in red) starts at a fixed location then turns left at a signalized junction, while another vehicle (in blue) crosses the intersection from the other side and a pedestrian crosses the street. (b) The ego car turns left and collides with an incoming car (in blue). (c) The ego car turns left and collides with a pedestrian crossing the street.	106
5.7 Three HD maps used by SCENORITA: Borregas Ave is a small map of a city block in Sunnyvale with 60 lanes and a total length of 3 km; San Mateo is a medium map with 1,305 lanes and a total length of 24 km; Sunnyvale is a large map consisting of 3061 lanes, with a total length of 107 km. . . .	108
5.8 An example of one scenario generated by SCENORITA with two reported violations: collision and hard braking. This image is obtained from Dreamview, the visual simulator of Apollo.	112
5.9 The total number of violations in tests reported by SCENORITA ⁺⁺ , SCENORITA ⁻ , and RANDOM	115

LIST OF TABLES

	Page
3.1 List of features <i>LibRARIAN</i> extracts from native binaries of Android apps along with their type and definition.	18
3.2 Heuristics used to search for unique per-library strings that contain version information	20
3.3 List of features <i>bin²sim</i> extracted from native binaries of Android apps along with their type and overall contribution factor, which measures the average percentage each feature contributes to the total similarity score	28
3.4 A list of libraries with reported <i>CVEs</i> found in our repository along with the number of distinct apps that were affected by a vulnerable library and the number of distinct apps containing a vulnerable version till now.	29
3.5 10 out of 14 popular apps from Google Play which include a vulnerable library that remained unchanged.	32
3.6 Combinations of 15 apps and particular vulnerable library versions they have contained, the date the vulnerability was publicly disclosed (<i>Vul announced</i>), the period between vulnerability disclosure and patch availability in days (i.e. Time-to-Release-Patch (<i>TTRP</i>)), and the total number of days elapsed before a fix was made (i.e. Time-to-Applied-Fix (<i>TTAF</i>))	34
3.7 Top 10 most negligent apps in terms of the average time to fix a vulnerable library	35
3.8 Top 10 most neglected vulnerable libraries in terms of the average time-to-fix	36
4.1 Statistics of Apollo and Autoware from GitHub	51
4.2 Additional Core Components with Significant Bugs	55
4.3 AV Sub-Components with Significant Bugs	55
4.4 Root Causes of Bugs in AV Systems	58
4.5 Symptoms of Bugs in AV Systems	60
4.6 Frequency of symptoms that each root cause of a bug may exhibit across Apollo and Autoware.	62
4.7 Frequency of bug occurrences for each AV component	66
4.8 Occurrences of bug symptoms in components of Apollo and Autoware	68
5.1 Comparing SCENORITA with the related work.	81
5.2 A list of <i>Domain-Specific Constraints</i> that <i>Scenario Generator</i> adheres to when creating driving scenarios.	89

5.3	The set of features, selected for each violation type, and used by the <i>Duplicate Violation Detector</i> to cluster similar violations together.	105
5.4	The number of all violations (All Viol.) reported by SCENORITA ⁺⁺ , SCENORITA ⁻ , and RANDOM, along with the total number of unique violations (Unique Viol.), and the percentage of duplicate violations eliminated (Elim. (%)). We highlight cells with the best reported results in grey.	113
5.5	The Average number of all violations (All Viol.) reported by three testing techniques (Test Tech.): SCENORITA ⁺⁺ , SCENORITA ⁻ and RANDOM on three maps (Borregas, San Mateo, and Sunnyvale), along with the average, minimum, and maximum number of unique violations (Unique Viol.), and the percentage of duplicate violations eliminated (Elim. (%)). We highlight cells with the best reported results in grey.	114
5.6	Ten case studies with reported violations generated by SCENORITA, along with a description of the scenarios. The videos corresponding to these case studies can be found in [1].	120
5.7	Efficiency of generated scenarios by SCENORITA ⁺⁺ , SCENORITA ⁻ and RANDOM	120

ACKNOWLEDGMENTS

The journey to my PhD degree has been one of the most challenging and arduous journeys in my entire life. With all its twists and turns, ups and downs, optimism and hopelessness, frustrating and exciting moments, and a PANDEMIC, I'm relieved that this unique and seemingly endless journey has come to an end. I'm extremely grateful for the support, guidance and love that I received from my advisors, mentors, colleagues, friends and family. I am where I'm at right now because of you, thank you so much!!!

First, and most importantly, I would like to thank my advisor Joshua Garcia, for believing in me and guiding me throughout my PhD. There have been numerous times when I seriously doubted my abilities and thought about dropping out, but his encouragement, empathy and understanding helped me power-through my self-doubt and hopelessness, and reach this final lap of my journey. Thank you, Josh! your expertise, knowledge, and understanding made me a better researcher today; I couldn't have chosen a better advisor! I'm so grateful.

I'm grateful to my PhD committee members, Sam Malek, and Qi Alfred Chen. Alfred, I'm fortunate that I had the chance to collaborate with you on multiple projects, and have weekly meetings with your group ever since you joined UC Irvine in 2018. My projects have benefited substantially from your insightful recommendations and feedback, thank you for being a part of my PhD journey. I would further like to thank Cristina Lopes and Ardalan Amiri Sani for serving on my advancement committee, as well as all of the faculty in the Institute for Software Research. I'd like to express my sincere gratitude to Mathias Payer, Sharad Mehrotra and Micah Sherr for guiding me in the early stages of my PhD.

I was fortunate enough to intern with the RISE group at Microsoft Research. I thank Shan Lu and Markus Kuppe for being an essential part of my research project at Microsoft. I'm especially grateful to my amazing manager Madan Musuvathi. Thank you, Madan, for believing in me and giving me the opportunity to work on one of the most exciting projects I worked on thus far. I particularly appreciate your mentorship, words of wisdom and advice that you provided me with throughout my internship. I learned a great deal from you and I'm forever grateful for having the opportunity to work with you.

Special thanks to my fellow lab mates in the SORA and SEAL group—Yuqi, Yuntianyi, Xiafa, Hongyu, Arda, Tuan, Jessy, Aziz, Navid, and Forough. I'll always remember our first lab in DBH and our endless trips to the Nespresso machine. I'm extremely thankful to Negar whom I got to know better when I moved to the Software Engineering department. I still remember how she greeted me, showed me around the lab, and made me feel so comfortable. Negar, I'm grateful for your support and love! I'm so lucky to have you as a close friend.

Dr. Shiva Sarabi, talking to you every week is what I looked forward to the most. You really helped me through one of the toughest, most stressful years in my life. Thank you for listening to me, supporting me, and guiding me through the hard days. I am forever indebted to you.

I could not have made it through my PhD program, and a PANDEMIC, without being surrounded by an amazing group of friends. I'm enormously grateful to Efi, Martha, Maruf, Mayara, Pedro and Sameera, thank you for making my PhD experience memorable and enjoyable. I'm particularly grateful for meeting one of the most amazing, wise, and kind people during my time in UC Irvine, my best friend Primal. Thank you for being there for me during the ups and downs, the good and bad, the serious and silly moments. I looked forward to our daily "cowalks" and our non-stop conversations about life. I'm so lucky to call you a close friend.

I am grateful to my parents, Abdullah and Helah, my brothers: Yahya, Hamad, Yossef, and Ibrahim for their constant love and support. I'm specifically grateful to my big sister and lovely cousin, Ghada for her understanding, never-ending encouragement and for always being there for me. Your support has meant to me more than you can possibly realize! Thank you, cousin!

I have to thank my soul mate, my lifetime companion and my best friend, Hadeel. Hadeel, you have proven that true friendship can survive distance and time constraints. You live on the opposite side of the world, yet you are the person that I talk to and share everything with the most. I'm so lucky to share this journey with you! Our daily 30-min voice messages, us planning our next adventures together, and sharing all the exciting and silly details about our lives brighten my days. I love you with all my heart!

Finally, I'm deeply indebted to my long-term partner, Francisco, who has been a constant source of strength, support, patience, and motivation for me throughout this entire experience. Thank you for the little things you've done, like bringing me "care packages" to snack on when I work late on my papers; for always planning fun things to do on the weekend to get my mind off work; for being my biggest cheerleader and my best friend. I am truly blessed to have you as my partner in this dance called "life". It's You and Me Babe, I love you so much!

The text in Chapter 3 of this dissertation is reprinted, with permission, from Arda Ünal, Mathias Payer and Joshua Garcia as it appears in "Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code", 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021 [51].

The text in Chapter 4 of this dissertation is reprinted, with permission, from Joshua Garcia, Yang Feng, Junjie Shen, Yuan Xia, and Qi Alfred Chen as it appears in "A comprehensive study of autonomous vehicle bugs", In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE), 2020 [100].

VITA

Sumaya Almanee

EDUCATION

Doctor of Philosophy in Software Engineering	2022
University of California, Irvine	<i>Irvine, CA</i>
Bachelor of Science in Computer Science	2011
King Saud University	<i>Riyadh, Saudi Arabia</i>

RESEARCH EXPERIENCE

Research Intern	Jan. 2022 – Aug. 2022
Microsoft Research	<i>Redmond, WA</i>
Graduate Research Assistant	Sept. 2016 – Aug. 2022
University of California, Irvine	<i>Irvine, California</i>
Visiting Researcher	Sept. 2015 – Aug. 2016
Georgetown University	<i>Washington, DC</i>

TEACHING EXPERIENCE

Teaching Assistant	Sept. 2016 – Aug. 2017
University of California, Irvine	<i>Irvine, CA</i>

REFEREED JOURNAL PUBLICATIONS

scenoRITA: Generating Diverse, Fully-Mutable, Safety-Critical and Motion Sickness-Inducing Scenarios for Autonomous Vehicles **2022**

Accepted pending major revision to appear in IEEE Transactions on Software Engineering (TSE 2022)

OBSCURE: Information-Theoretically Secure, Oblivious, and Verifiable Aggregation Queries on Secret-Shared Outsourced Data **March 2020**

IEEE Transactions on Knowledge and Data Engineering (TKDE 2020)

REFEREED CONFERENCE PUBLICATIONS

Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code **May 2021**

In the 43rd International Conference on Software Engineering (ICSE 2021)

A Comprehensive Study of Autonomous Vehicle Bugs **May 2020**

In the 42nd International Conference on Software Engineering (ICSE 2020)

Obscure:Information-Theoretic Oblivious and Verifiable Aggregation Queries **Sept. 2019**

In the 45th International Conference on Very Large Data Bases (VLDB 2019)

REFEREED WORKSHOP PUBLICATIONS

SemIoTic: Bridging the Semantic Gap in IoT Spaces. **Nov. 2019**

In the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys 2019)

ABSTRACT OF THE DISSERTATION

Assuring Software Dependability of Smart Systems

By

Sumaya Almanee

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2022

Assistant Professor Joshua Garcia, Chair

Smart systems are software entities that carry out a set of operations on behalf of a user or another application with some degree of independence or autonomy [65, 156]. These systems employ some knowledge or representations of (1) a user’s goals or desires, and (2) the environment in which they act in to achieve these goals. Such an agent is a system situated in a technical or natural environment that senses some status from that environment and acts on it, changing part of its environment or influencing what it senses.

The main characteristics of smart systems [156] are *adaptive capacity*, indicating that such systems adapt as information changes, and they may resolve ambiguity and tolerate unpredictability; *learning capability*, implying that smart systems reason on data to create new information and use closed-loop feedback to learn from the output; *context awareness*, indicating that smart systems may identify and extract contextual elements such as syntax, time, location, etc; and *dynamic interactivity*, indicating that these systems can interact with users or other applications and cloud services to understand their goals and needs.

Smart systems address environmental, societal, and economic challenges like limited resources, climate change, and globalization. They are, for that reason, increasingly used in a large number of sectors such as transportation, healthcare, energy, safety, security, etc. Hence, the need for effective analysis and testing techniques for such systems has increased more than

ever.

This dissertation proposes to ensure the dependability (i.e., security, safety, and reliability) of smart systems by (1) analyzing bugs and vulnerabilities found in such systems and (2) developing tools to test and detect bugs and vulnerabilities in smart systems automatically.

Experiments conducted on real-world, open-source software applications corroborates the effectiveness and efficiency of our proposed approaches and their ability to ensure the dependability of software in smart systems.

Chapter 1

Introduction

Since the birth of the Internet in the late 1980s, we found ourselves living in two seemingly parallel worlds: One is the familiar physical world, and the other is this growing world of information. The convergence of advanced technologies and analytics, cloud computing, and smart systems allowed these two parallel worlds to collide, as the wave of technology—that started with personal computers—broke out into the world of physical things. The age of smart systems is becoming a reality as more products and services that we use every day—like search-engine applications, face recognition, smart cars, and phones—demonstrate smart and adaptive behavior. These smart systems incorporate sensing, actuation, and control functions to analyze and make decisions in a predictive or adaptive manner, thereby performing smart actions [156]. In most cases, the “smartness” of a system is attributed to autonomous operation based on closed-loop control, machine learning, and networking capabilities that enable the system to exhibit adaptive behavior [65]. Nowadays, smart systems sit at the intersection of humans and technology infrastructures, as they perform basic control operations for our technology infrastructure and interact with people to understand their needs and perform required actions.

Smart systems address environmental, societal, and economic challenges like limited resources, climate change, and globalization. They are, for that reason, increasingly used in a large number of sectors such as transportation, healthcare, energy, safety, security, etc. For example, in the automotive sector, smart systems integration will be a key enabler for pre-crash systems and predictive driver assistance features. Yet there are many challenges introduced by the rise of smart system technologies, such as safety, security, stability, scalability, efficiency, etc.

One example of such challenges is the safety of autonomous vehicles (AVs), a.k.a. self-driving cars or smart cars. Experts forecast that AVs will drastically impact society, particularly by reducing accidents [61]. However, crashes caused by AVs indicate that achieving this lofty goal remains an open challenge. Despite the fact that companies such as Tesla [27], Waymo [36], or Uber [33] have released prototypes of AVs with a high level of autonomy, they have caused injuries or even fatal accidents to pedestrians. For instance, an AV of Uber killed a pedestrian in Arizona back in 2018 [26].

Prior research has revealed a lack of standardized procedures to test AVs [120] and the inability of current approaches to effectively translate traditional software testing approaches into the space of AVs [124, 112]. A common practice for testing AV software lies in field operational tests, in which AVs are left to drive freely in the physical world. This approach is not only expensive and dangerous but also ineffective since it misses critical testing scenarios [116]. Virtual tests, where AVs are tested in software simulations, offer a far more efficient and safer alternative. While these tests provide an opportunity to automatically generate tests, they come with the key challenge of *systematically generating scenarios that expose AVs to safety-critical and motion sickness-inducing situations*.

Another challenge in smart systems is related to the improper handling of security updates, especially security updates in native libraries of smartphones. Vulnerabilities found in such libraries may propagate to host apps increasing the attack surface of such apps [160, 166]. The security implications in native libraries are critical for the following reasons: (1) app

developers add native libraries but do not update them, either due to concerns over regressions arising from such updates, prioritizing new functionality over security, deadline pressures, and other forms of negligence (such as a lack of tracking library dependencies and their patches) that results in outdated or vulnerable native libraries remaining in new versions of apps; (2) native libraries are susceptible to memory vulnerabilities (e.g., buffer overflow attacks) that are very difficult to exploit with managed code of Android apps, i.e., Dalvik code; (3) contrary to previous studies [83, 187], native libraries are currently used pervasively in top mobile apps.

The aforementioned challenges highlight the importance of ensuring software dependability in smart systems such as safety, security, reliability, maintainability, etc. In the remainder of this section, I will explain the related background and challenges of smart systems software in more detail.

1.1 Security Updates in Android Apps’ Native Code

Third-party libraries are an integral part of mobile apps. Android developers opt for third-party libraries due to their convenience and re-usability, since utilizing them saves time and effort and allows developers to avoid re-implementing functionality. Furthermore, native libraries have become more prevalent in recent Android applications (“apps”), especially social networking and gaming apps. These two app categories—ranked among the top categories on Google Play—require special functionality such as 3D rendering, or audio/video encoding/decoding [93, 135, 173, 150, 164]. These tasks tend to be resource-intensive and are, thus, often handled by native libraries to improve runtime performance.

The ubiquity of third-party libraries in Android apps increases the attack surface [160, 166] since host apps expose vulnerabilities propagated from these libraries [110, 163]. Another series of previous work has studied the outdatedness and updateability of third-party *Java libraries* in Android apps [79, 58], with a focus on managed code of such apps (e.g., Java

or Dalvik code). However, these previous studies do not consider *native libraries* used by Android apps.

We argue that security implications in native libraries are even more critical for three main reasons. First, app developers add native libraries but neglect to update them. The reasons for this may include concerns over regressions arising from such updates, prioritizing new functionality over security, deadline pressures, or lack of tracking library dependencies and their security patches. This negligence results in outdated or vulnerable native libraries remaining in new versions of apps. Second, native libraries are susceptible to memory vulnerabilities (e.g., buffer overflow attacks) that are straightforward to exploit. Third, and contrary to studies from almost 10 years ago [83, 187], native libraries are now used pervasively in mobile apps. To illustrate this point, we analyzed the top 200 apps from Google Play between Sept. 2013 and May 2020. We obtained the version histories of these apps from *AndroZoo* [50] totaling 7,678 versions of those 200 top free apps. From these apps, we identified 66,684 native libraries in total with an average of 11 libraries per app and a maximum of 141 for one version of *Instagram*.

To better understand the usage of third-party native libraries in Android apps and its security implications, a *longitudinal study* must be conducted; this study investigates (1) the prevalence of vulnerabilities in native libraries in the top apps, and (2) the rate at which app developers apply patches to address vulnerabilities in native binaries. A core challenge we face in this study is the identification of libraries and their versions, since developers often rename or modify libraries, making their identification challenging. The previous challenge requires an approach that accurately *identifies native libraries and their versions* as found in Android apps. Chapter 3 describes our approach to identifying security vulnerabilities in Android apps in more detail.

1.2 Analysis of Bugs in Autonomous Vehicles

Autonomous Vehicles (AV) technology leverages advanced sensing and networking technologies (e.g., camera, LiDAR, RADAR, GPS, DSRC, 5G, etc.) to enable safe and efficient driving without human drivers. Although still in its infancy, AV technology is becoming increasingly common and could radically transform our transportation system and by extension, our economy and society. As a result, there is tremendous global enthusiasm for research, development, and deployment of autonomous vehicles (AVs), e.g., self-driving taxis and trucks from Waymo and Baidu.

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 [80], machine learning libraries [113, 181, 167], concurrency bugs [126, 138], performance bugs [114, 159], and error-handling bugs [169, 73, 66]. None of these studies have focused on bugs in AV software systems. Chapter 4 presents a comprehensive study of bugs in AVs; providing a classification of root causes and symptoms of bugs, and the AV components affected by these bugs.

1.3 Test Generation for Autonomous Vehicles

The current practice for testing AVs uses virtual tests—where AVs are tested in software simulations—since they offer a more efficient and safer alternative compared to field op-

erational tests. Specifically, search-based approaches are used to find particularly critical situations. While these tests provide an opportunity to generate tests automatically, they come with the key challenge of *systematically generating scenarios that expose AVs to safety-critical and motion sickness-inducing situations*.

Additionally, previous work on AV software testing uses a highly limited number of test oracles for ensuring safety and no oracles for assessing motion sickness-inducing movement of an AV. For example, state-of-the-art AV testing approaches (*AC3R*, *AV-Fuzzer*, *AsFault* and Abdessalem et al. [60, 47]) use only two oracles for checking if (1) the ego car reaches its final expected position while avoiding a crash (i.e., collision detection) and (2) if a vehicle drives off the road (i.e., off-road detection); while completely ignoring rider comfort and motion sickness. Research has shown that a rider’s discomfort increases when a human is a passenger rather than a driver—with up to one-third of Americans experiencing motion sickness, according to the National Institutes of Health (NIH) [14, 23, 128].

We address the previous challenges by (1) proposing a test generation approach that automatically generates *valid* and *effective* driving scenarios, (2) we utilize 5 test oracles to determine both safety and motion sickness-inducing violations, and (3) we introduce a novel technique to identify and eliminate duplicate tests for autonomous vehicles. Chapter 5 describes our automatic test generation approach for AVs in more detail.

1.4 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 presents the research problem, three research hypotheses, and the scope of this thesis. Chapter 3 introduces *LibRARIAN*, an approach that accurately identifies native libraries and their versions as found in Android apps along with a large-scale, longitudinal study that tracks security vulnerabilities in native libraries used in apps over 7 years[51]. Chapter 4 presents a comprehensive study of bugs in AV systems which consists of a classification of root causes and symptoms of bugs, and

the AV components these bugs may affect. Finally, Chapter 5 describes SCENORITA, a test generation framework that aims to find safety and motion sickness-inducing violations in the presence of an evolving traffic environment.

The following is the list of my research projects and publications:

- **Sumaya Almanee**, Xiafa Wu, Yuqi Huai, Qi Alfred Chen, and Joshua Garcia. "*sceno-RITA: Generating Diverse, Fully-Mutable, Safety-Critical and Motion Sickness-Inducing Scenarios for Autonomous Vehicle*". Accepted pending Major Revision to appear in IEEE Transactions on Software Engineering (TSE 2022).
- **Sumaya Almanee**, Arda Unal, Mathias Payer, and Joshua Garcia. "*Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code*". In the 43rd International Conference on Software Engineering (ICSE 2021).
- Joshua Garcia, Yang Feng, Junjie Shen, **Sumaya Almanee**, Yuan Xia, and Qi Alfred Chen. "*A Comprehensive Study of Autonomous Vehicle Bugs*". In the 42nd International Conference on Software Engineering (ICSE 2020).
- Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwat, Shantanu Sharma, **Sumaya Almanee**. "*Obscure:Information-Theoretically Secure, Oblivious, and Verifiable Aggregation Queries on Secret-Shared Outsourced Data*". IEEE Transactions on Knowledge and Data Engineering (TKDE 2020).
- Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwat, Shantanu Sharma, **Sumaya Almanee**. "*Obscure:Information-Theoretic Oblivious and Verifiable Aggregation Queries*". In the 45th International Conference on Very Large Data Bases (VLDB 2019).
- **Sumaya Almanee**, Georgios Bouloukakis, Daokun Jiang, Sameera Ghayyur, Dhruvajyoti Ghosh, Peeyush Gupta, Yiming Lin, Sharad Mehrotra, Primal Pappachan, Eun-Jeong Shin, Nalini Venkatasubramanian, Guoxi Wang, Roberto Yus." *SemIoTic: Bridg-*

ing the Semantic Gap in IoT Spaces". In the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys 2019).

Chapter 2

Research Problem

2.1 Problem Statement

Smart systems sit at the intersection of humans and technology infrastructures, as they perform basic control operations for our technology infrastructure and interact with people to understand their needs and goals. The challenges caused by the lack of techniques to ensure the dependability of software in smart systems can be summarized as follow: “*Smart systems are increasingly used in a large number of sectors such as transportation, healthcare, energy, safety, and security. Hence, the need for ensuring software dependability in such systems has increased more than ever. More specifically, there is a demand for enforcing measures for the **security**, **reliability**, and **safety** of software found in such systems. For example, a comprehensive understanding and analysis of the nature of bugs and security vulnerabilities found in smart systems may lead to a better creation of bug detection techniques and testing tools for such systems. Additionally, effective and valid test generation techniques are required to assess the behavior of smart systems (such as smart cars) under different conditions. Unfortunately, there is a lack of appropriate frameworks and techniques among the state-of-the-art works which effectively ensures software dependability in smart systems.*”

2.2 Research Hypothesis

2.2.1 Software Security in Smart Systems

Developers rarely perform code reviews on third-party dependencies [79], despite the fact that prior work [160, 166, 110, 163] has shown that third-party dependencies are often a source of software vulnerabilities. More specifically, native libraries in Android apps are often neglected, and as a result, remain outdated with unpatched security vulnerabilities years after patches become available. Unfortunately, none of the previous studies [79, 58] examined native third-party libraries in Android apps nor did they look at the security impact of vulnerable libraries or whether these vulnerabilities are on the attack surface.

Hypothesis 1: *An automated and efficient approach can be devised to accurately identify native libraries and their versions as found in Android apps.*

This tool enables me to achieve the second objective in my research which is to:

Hypothesis 2: *Conduct a large-scale study to examine the prevalence of security vulnerabilities in Android’s native libraries can be conducted.*

Work Progress: To test these hypotheses, I created an approach called *LibRARIAN* that, given an unknown binary, identifies (i) the library it implements and (ii) its version. We then conducted a large-scale, longitudinal study that tracks security vulnerabilities in native libraries used in apps for over 7 years. We discovered 53/200 popular apps (26.5%) with vulnerable versions with known CVEs between Sept. 2013 and May 2020, with 14 of those apps remaining vulnerable. We found that app developers took, on average, 528.71 ± 40.20 days to apply security patches, while library developers release a security patch after 54.59 ± 8.12 days—a 10 times slower rate of update.

2.2.2 Software Reliability in Smart Systems

One of the objectives of ensuring software reliability in smart systems is identifying the root cause of failures that might occur in such systems. Previous empirical studies have investigated bug characteristics in a variety of domains including numerical software libraries [80], machine learning libraries [113, 181, 167], concurrency bugs [126, 138], performance bugs [114, 159], and error-handling bugs [169, 73, 66]. None of these studies have focused on bugs in AV software systems. 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.

Hypothesis 3: *A comprehensive study of AV bugs can be conducted to identify the root causes and symptoms of bugs, and the AV components affected by these bugs.*

Work Progress: To test this hypothesis, we conducted a comprehensive study of bugs in two major AV platforms, Apollo and Autoware [15, 13]. We investigated the bugs found in the previous two AV platforms in addition to their root causes, their impacts, and the type of AV components they affect. We have studied a total of 499 AV bugs from 16,851 commits across Apollo and Autoware repositories, of which we have identified 13 root causes and 20 symptoms across 18 AV components.

2.2.3 Software Safety in Smart Systems

The current practice for testing AVs uses virtual tests—where AVs are tested in software simulations—since they offer a more efficient and safer alternative compared to field op-

erational tests. Specifically, search-based approaches are used to find particularly critical situations. While these tests provide an opportunity to generate tests automatically, they come with the key challenge of *systematically generating scenarios that expose AVs to safety-critical and motion sickness-inducing situations*. Prior work [130, 96, 97, 67] ignores the challenge of ensuring the creation of valid obstacle trajectories, reducing their *effectiveness at generating driving scenarios with unique violations*. Moreover, previous work uses a highly limited number of test oracles for ensuring safety and no oracles for assessing motion sickness-inducing movement of an AV.

Hypothesis 4: *A search-based test generation framework can be devised for smart cars to find diverse, fully-mutable, safety and motion sickness-inducing violations in the presence of an evolving traffic environment.*

Work Progress: To test this hypothesis, I have developed SCENORITA, a novel search-based testing framework that generates driving scenarios that expose AV software to 3 types of safety-critical and 2 types of motion sickness-inducing violations. SCENORITA reduces duplicate scenarios, allows fully mutable obstacles with valid and modifiable obstacles trajectories, and follows domain-specific constraints obtained from authoritative sources. Using this approach, we found a total of 1,026 unique safety and comfort violations including 386 collisions, 21 speed violations, 291 unsafe lane changes, 132 fast acceleration violations, and 196 hard-braking violations.

Chapter 3

An Empirical Study of Security Updates in Android Apps' Native Code

Android apps include third-party native libraries to increase performance and to reuse functionality. Native code is directly executed from apps through the Java Native Interface or the Android Native Development Kit. Android developers add precompiled native libraries to their projects, enabling their use. Unfortunately, developers often struggle or simply neglect to update these libraries in a timely manner. This results in the continuous use of outdated native libraries with unpatched security vulnerabilities years after patches became available.

To further understand such phenomena, we study the security updates in native libraries in the most popular 200 free apps on Google Play from Sept. 2013 to May 2020. A core difficulty we face in this study is the identification of libraries and their versions. Developers often rename or modify libraries, making their identification challenging.

We create an approach called *LibRARIAN* (**LibR**Ary **veR**sion **IdentificAtioN**) that accu-

rately identifies native libraries and their versions as found in Android apps based on our novel similarity metric *bin²sim*. *LibRARIAN* leverages different features extracted from libraries based on their metadata and identifying strings in read-only sections.

We discovered 53/200 popular apps (26.5%) with vulnerable versions with known CVEs between Sept. 2013 and May 2020, with 14 of those apps remaining vulnerable. We find that app developers took, on average, 528.71 ± 40.20 days to apply security patches, while library developers release a security patch after 54.59 ± 8.12 days—a 10 times slower rate of update.

3.1 Introduction

Third-party libraries are convenient, reusable, and form an integral part of mobile apps. Developers can save time and effort by reusing already implemented functionality. Native third-party libraries are prevalent in Android applications (“apps”), especially social networking and gaming apps. These two app categories—ranked among the top categories on Google Play—require special functionality such as 3D rendering, or audio/video encoding/decoding [93, 135, 173, 150, 164]. These tasks tend to be resource-intensive and are, thus, often handled by native libraries to improve runtime performance.

The ubiquity of third-party libraries in Android apps increases the attack surface [160, 166] since host apps expose vulnerabilities propagated from these libraries [110, 163]. Another series of previous work has studied the outdatedness and updateability of third-party *Java libraries* in Android apps [79, 58], with a focus on managed code of such apps (e.g., Java or Dalvik code). However, these previous studies do not consider *native libraries* used by Android apps.

We argue that security implications in native libraries are even more critical for three main reasons. First, app developers add native libraries but neglect to update them. The reasons for this may include concerns over regressions arising from such updates, prioritizing new

functionality over security, deadline pressures, or lack of tracking library dependencies and their security patches. This negligence results in outdated or vulnerable native libraries remaining in new versions of apps. Second, native libraries are susceptible to memory vulnerabilities (e.g., buffer overflow attacks) that are straight-forward to exploit. Third, and contrary to studies from almost 10 years ago [83, 187], native libraries are now used pervasively in mobile apps. To illustrate this point, we analyzed the top 200 apps from Google Play between Sept. 2013 and May 2020. We obtained the version histories of these apps from *AndroZoo* [50] totaling 7,678 versions of those 200 top free apps. From these apps, we identified 66,684 native libraries in total with an average of 11 libraries per app and a maximum of 141 for one version of *Instagram*.

To better understand the usage of third-party native libraries in Android apps and its security implications, we conduct a longitudinal study to identify vulnerabilities in third-party native libraries and assess the extent to which developers update such libraries of their apps. In order to achieve this, we make the following research contributions:

- We construct a novel approach, called *LibRARIAN* (**LibR**Ary ve**R**sion Identific**A**tio**N**) that, given an unknown binary, identifies (i) the library it implements and (ii) its version. Furthermore, we introduce a new similarity-scoring mechanism for comparing native binaries called *bin²sim*, which utilizes 6 features that enable *LibRARIAN* to distinguish between different libraries and their versions. The features cover both metadata and data extracted from the libraries. These features represent elements of a library that are likely to change between major, minor, and patch versions of a native library.
- We conduct a large-scale, longitudinal study that tracks security vulnerabilities in native libraries used in apps over 7 years. We build a repository of Android apps and their native libraries with the 200 most popular free apps from Google Play totaling 7,678 versions gathered between the dates of Sept. 2013 and May 2020. This repository further contains 66,684 native libraries used by these 7,678 versions.

Prior work [143, 133, 111, 49] has measured the similarity between binaries. However, these approaches identify semantic similarities/differences between binaries at the *function-level*, with the goal of identifying malware. *LibRARIAN*, orthogonally, is a syntactic-based tool which computes similarity between two benign binaries (at the file-level) with the goal of identifying library versions with high scalability.

We utilize *LibRARIAN* and our repository to study (1) *LibRARIAN*’s accuracy and effectiveness, (2) the prevalence of vulnerabilities in native libraries in the top 200 apps, and (3) the rate at which app developers apply patches to address vulnerabilities in native binaries. The major findings of our study are as follows:

- For our ground truth dataset which contains 46 known libraries with 904 versions, *LibRARIAN* correctly identifies 91.15% of those library versions, thus achieving a high identification accuracy.
- To study the prevalence of vulnerabilities in the top 200 apps in Google Play, we use *LibRARIAN* to examine 53 apps with vulnerable versions and known CVEs between Sept. 2013 and May 2020. 14 of these apps remain vulnerable and contain a wide-range of vulnerability types—including denial of service, memory leaks, null pointer dereferences, or divide-by-zero errors. We further find that libraries in these apps, on average, have been outdated for 859.17 ± 137.55 days. The combination of high severity and long exposure of these vulnerabilities results in ample opportunity for attackers to target these highly popular apps.
- To determine developer response rate of applying security fixes, we utilize *LibRARIAN* to analyze 40 apps, focusing on popular third-party libraries (those found in more apps) with known CVEs such as *FFmpeg*, *GIFLib*, *OpenSSL*, *WebP*, *SQLite3*, *OpenCV*, *Jpeg-turbo*, *Libpng*, and *XML2*, between Sept. 2013 and May 2020.

We find that app developers took, on average, 528.71 ± 40.20 days to apply security patches, while library developers release a security patch after 54.59 ± 8.12 days—a 10 times slower

rate of update. These libraries that tend to go for long periods without being patched affect highly popular apps with billions of downloads and installs.

- We make our dataset, analysis platform, and results available online to enable reusability, reproducibility, and others to build upon our work [134].

3.2 *LibRARIAN*

Figure 3.1 shows the overall workflow of *LibRARIAN*. *LibRARIAN* identifies unknown third-party native libraries and their versions (*Unknown Lib Versions*) by (1) extracting features that distinguish major, minor, and patch versions of libraries that are stable across platforms regardless of underlying architecture or compilation environments; (2) comparing those features against features from a ground-truth dataset (*Known Lib Versions*) using a novel similarity metric, bin^2sim ; and (3) matching against strings that identify version information of libraries extracted from *Known Lib Versions*, which we refer to as *Version Identification Strings*. In the remainder of this section, we describe each of these three major steps of *LibRARIAN*.

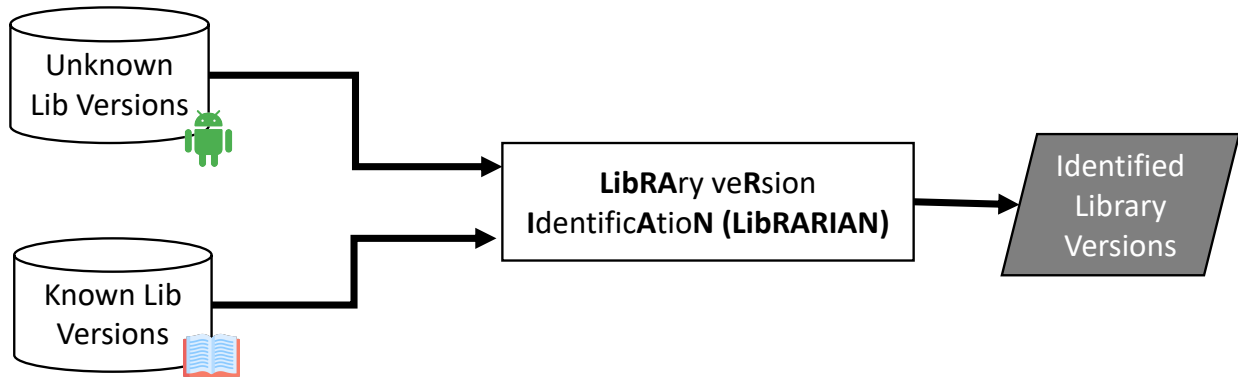


Figure 3.1: *LibRARIAN* identifies versions of native binaries from Android apps by using our bin^2sim similarity-scoring technique to compare known (ground-truth dataset) and unknown versions of native binaries.

3.2.1 Feature Vector Extraction

Our binary similarity detection is based on the extraction of features from binaries combining both metadata found in Executable and Linkable Format (ELF) files as well as identifying features in different binary sections of the library. All shared libraries included in Android apps are compiled into ELF binaries. Like other object files, ELF binaries contain a symbol table with externally visible identifiers such as function names, global symbols, local symbols, and imported symbols. This symbol table is used (1) during loading and linking and (2) by binary analysis tools [102] (e.g., `objdump`, `readelf`, `nm`, `pwntools`, or `angr` [162]) to infer information about the binary.

Feature Type	Name	Definition
Metadata	Exported Globals Imported Globals Exported Functions	Externally visible variables, i.e., they can be accessed externally. Variables from other libraries that are used in this library. Externally visible functions, i.e., functions that can be called from outside the library.
	Imported Functions Dependencies	Functions from other libraries that are used in this library. The library dependencies that are automatically loaded by the ELF object
Data	Version Identification Strings	Flexible per-library version strings (e.g., “libFoo-1.0.2” matched to strings in the <i>.rodata</i> section of an ELF object)

Table 3.1: List of features *LibRARIAN* extracts from native binaries of Android apps along with their type and definition.

To distinguish between different libraries and their versions, we need to identify *differencing features*. To that end, we define a set of six features inherent to versions and libraries. Five features represent ELF metadata, these features are used to compute the similarity score between two binaries as described in Section 3.2.2, hence, we refer to these features as *Metadata Features*. Orthogonally, we leverage strings extracted from the *.rodata* section of an ELF object, which we refer to as *Version Identification Strings*. This feature complements the similarity score from the first set of features. We either use it to verify the correctness of the version or as a fallback if the similarity to existing binaries in our ground-truth dataset is low (see Section 3.2.3).

Table 3.1 shows the list of all *LibRARIAN* features. The features include: (i) five *Metadata*

Features based on exported and imported functions, exported and imported globals, and library dependencies; and (ii) one *Data Feature* which is applied as a second factor to either substitute the *Metadata Features*, in case the reported similarity score is low, or to confirm the reported score. These 6 features represent the code elements of a library that would be expected to change based on a versioning scheme that distinguishes major, minor, and patch versions of a library. Furthermore, these features are stable across platforms regardless of the underlying architecture or compilation environments. We did not include code features (e.g., control-flow and data-flow features) as they are extremely volatile and change between compilations and across architectures. Binary similarity matching is a hard open problem: While recent work has made progress regarding accuracy [85, 178, 143, 133, 111, 49, 108, 81], the majority of algorithms have exponential computation cost relative to the code size and are infeasible for large-scale studies.

We built a dataset of heuristics by inspecting the binaries in our ground-truth dataset. We developed scripts to process the data in the *.rodata* sections extracted during feature processing and search for unique per-library strings that contain version information. For example, *FFmpeg* version info is found when applying the regex `ffmpeg-([0-9]\.)*[0-9]` or `FFmpeg version([0-9]\.)*[0-9]`. Table 3.2 shows our list of extracted version heuristics. Each version heuristic can be produced automatically by constructing regular expressions from strings in *.rodata* sections of binaries in our ground-truth dataset. For example, if the string “*libFoo-1.0.2*” is found in version 1.0.2 of *libFoo*, *LibRARIAN* uses a regular expression replacing the numeric suffix of the string with an appropriate pattern (e.g., `libFoo-[0-9]+(\.[0-9])*`).

We deliberately exclude any metadata or identifying strings for symbols that are volatile across architectures or build environments like compiler version, relocation information (and types), or debug symbols. *LibRARIAN*’s accuracy results in Section 3.3.1 demonstrate that our selected set of features suffice to distinguish between different versions of libraries.

Library Name	Extracted Heuristics
Jpeg-turbo	Jpeg-turbo version 1(\.[0-9]{1,})*
FFmpeg	ffmpeg-(\.[0-9]{1,})* FFmpeg version (\.[0-9]{1,})*
Firestore	Firestore C++ [0-9]+(\.[0-9]{1,})*
Libavcodec	Lavc5[0-9](\.[0-9]{1,})*
Libavfilter	Lavf5[0-9](\.[0-9]{1,})*
Libpng	Libpng version 1(\.[0-9]{1,})*
Libglog	glog-[0-9]+(\.[0-9]{1,})*
Libvpx	WebM Project VP(\.[0-9]{1,})*
OpenCV	General configuration for OpenCV [0-9]+(\.[0-9]{1,})* opencv-[0-9]+(\.[0-9]{1,})*
OpenSSL	openssl-1(\.[0-9]{1,})*[a-z] ~OpenSSL 1(\.[0-9]{1,})*[a-z]
Speex	speex-(\.[0-9]{1,})*
SQLite3	~3\.[0-9]{1,}(\.[0-9]{1,})*[a-z]
Unity3D	(\.[0-9]{1,})+([a-z][0-9]{1,}) Expected version:(\.[0-9]{1,})*
Vorbis	Xiph.Org Vorbis 1.(\.[0-9]{1,})*
XML2	GITv2.[0-9]+(\.[0-9]{1,})*

Table 3.2: Heuristics used to search for unique per-library strings that contain version information

The implementation leverages angr’s [162] ELF parser which already is platform independent. Our extraction platform recovers all metadata from the ELF symbol tables and, if available, searches for string patterns in the comment and read-only sections. Our filters remove platform specific information and calls to standard libraries (e.g., C++ ABI calls, vectors, or other data structures). The current implementation covers x86-64, x86, ARM, and ARM64 binaries—which are all platforms we observed in our evaluation. We accommodate for architecture differences in two ways: First, we remove architecture noise in feature vectors (e.g., symbols that are only used in one architecture); and second, we collect, if available, binaries for the different architectures.

The feature extraction compiles all recovered information as a dictionary into a JSON file. The dictionary contains arrays of strings for each of the features mentioned above plus additional metadata to identify the library and architecture.

3.2.2 Similarity Computation

LibRARIAN’s similarity computation, which we refer to as bin^2sim , leverages the five *Metadata Features* when computing the similarity scores between an app binary and our ground-truth dataset. bin^2sim is based on the *Jaccard coefficient*, and is used to determine the similarity between feature vectors. bin^2sim allows *LibRARIAN* to account for addition

or removal of features between different libraries and versions. Given two binaries b_1 and b_2 with respective feature vectors FV_1 and FV_2 , bin^2sim computes the size of the intersection of FV_1 and FV_2 (i.e., the number of common features) over the size of the union of FV_1 and FV_2 (i.e., the number of unique features):

$$bin^2sim(FV_1, FV_2) = \frac{|FV_1 \cap FV_2|}{|FV_1 \cup FV_2|} \in [0, 1] \quad (3.1)$$

The similarity score is a real number between 0 and 1, with a score of 1 indicating identical features, a score of 0 indicating no shared features between the two libraries, and a fractional value indicating a partial match. Due to the volatility of the similarity score, filtering noise such as platform-specific details as mentioned in the previous section is essential for the accuracy of our approach.

LibRARIAN counts an unknown library instance from *Unknown Lib Versions* as matching a known library version if its bin^2sim is above 0.85. This threshold was determined experimentally and works effectively as our evaluation will demonstrate (see Section 5.5). If bin^2sim results in the same value above the threshold for multiple known binaries, *LibRARIAN* tries obtaining an exact match between one of the known binaries and the unknown binary by using their hash codes to determine the unknown binary’s version.

A low similarity score might result from modifications made by app developers to the original third-party library which results in the removal or addition of specific features. From our experience, removal of features from the original library is common among mobile developers and is likely driven by the need to reduce the size of the library and the app as much as possible. For example, we observed that the WebP video codec library is often deployed without encoding functionalities to reduce binary size. Some size optimization techniques require choosing needed modules from a library and leaving the rest, stripping the resulting

binary, and modifying build flags. Another factor that reduces similarity as measured by the Jaccard coefficient is that certain architectures tend to export more features as compared to others. For instance, 32-bit architectures such as armeabi-v7a and x86 export more features compared to arm64-v8a and x86_64.

3.2.3 Version Identification Strings

For libraries where *LibRARIAN* reports low similarity scores (e.g., some libraries like *RenderScript* or *Unity* only export a single function ¹), these five features fail to provide sufficient information about the underlying components in a library. If libraries only export one or a few functions, the similarity metrics have a hard time distinguishing between different libraries. We therefore extend the features with strings that uniquely identify the library. Such strings are often version strings. Based on extracted flexible per-library heuristics from our ground-truth dataset (see Table 3.2), we heuristically identify exact library versions and increase overall accuracy. For libraries with high similarity scores, we use these library heuristics to confirm the correct version.

To identify binaries with low similarity scores, we leverage *Version Identification Strings*, which is the set of extracted per-library version strings. For example, say a library version *lv* extracted from app *a* had a similarity score of 0.3 when compared with *OpenCV-2.4.11* using *Metadata Features*. Given the low score, we search the *Version Identification Strings* feature for specific keywords such as `General configuration for OpenCV *.*.*` or `opencv-*.*.*`. Where the asterisk represents the versioning scheme of *OpenCV* library.

Our feature extraction process logs all strings (arrays of more than 3 ASCII printable characters ending with a 0 byte) from the *.rodata* section alongside the other features. As libraries commonly have large amounts of read-only string data that frequently changes,

¹These libraries are “stripped” and hide all functionality internally. The single exported function takes a string as parameter which corresponds to the target function and they dispatch to internal functionality based on this string.

we cannot use this data directly as a feature (due to the low overlap resulting in low similarity). By processing the *.rodata* from our ground-truth dataset and clustering the data, we extract common version identifiers and version strings. We then translate them into regular expressions that allow us to match versions for different libraries.

3.3 Evaluation

To assess the prevalence of vulnerable native libraries for Android, we answer the following three research questions:

RQ1: *Accuracy and effectiveness of LibRARIAN.* Can *LibRARIAN* accurately and effectively identify versions of native libraries? How does *LibRARIAN* compare against state-of-the-art native-library version identification? How effective are *LibRARIAN*’s feature types at identifying versions of native libraries?

RQ2: *Prevalence of outdated libraries.* How prevalent are vulnerabilities in native libraries of Android apps?

RQ3: *Patch response time.* After a vulnerability is reported for a third-party library, how quickly do developers apply patches?

To supplement the aforementioned RQs, we conducted a detailed case study on a vulnerable app (Section 3.3.4), providing practical insight into vulnerabilities in third-party libraries and possible exploits.

To answer these research questions, we analyze the top 200 apps in Google Play over several years. We track the version history of these apps from AndroZoo [50], a large repository of over 11 million Android apps. Our repository contains app metadata including the app name, release dates, and native binaries.

Note that Google Play unfortunately restricts lists to 200 apps. Overall, we collected 7,678 instances, where each instance is a version of the 200 top apps from Google Play.

We determined that 145 out of 200 (72.50%) of the distinct apps in our repository contain at least one native library, i.e., 5,852 out of 7,678 (76.21%) of the total apps in our database. There are a total of 66,684 libraries in the form of *.so* files, i.e., shared library files, in our repository with an average of 11 libraries per app and a maximum of 141 for one version of *Instagram*. In fact, *Instagram*—for which we collected 184 versions since Dec. 2013—contains a total of 6,677 *.so* files.

We run *LibRARIAN* on a machine with 2 AMD EPYC 7551 32-Core CPUs and 512GB of RAM running Ubuntu 18.04. The average number of features in the extracted feature vectors is 2,116.86 features. Some outliers such as *libWaze* and *libTensorflow* reach up to 79,581 features. This shows that the set of third-party native libraries in our repository is diverse, some of them are very complex and offer a large number of functionalities. Generating feature vectors is quick and generally takes a few seconds per library. The most complex library, *libTensorflow* takes 4 min and 38 sec to analyze. We found that, out of 7,253 binaries for which *LibRARIAN* inferred their versions, the average runtime for library version detection is 118.19 seconds—with a minimum of 97 seconds and a maximum of 224 seconds.

3.3.1 RQ1: Accuracy and Effectiveness

To determine if *LibRARIAN* accurately and effectively identifies native library versions from Android apps, we assess *LibRARIAN* in three scenarios. For the first scenario, we compare its accuracy with *OSSPolice*, the state-of-the-art technique for identifying versions of native binaries for Android apps. For the second scenario, we assess *LibRARIAN* on a larger dataset for which *OSSPolice* could not be applied and, thus, evaluate *LibRARIAN*’s accuracy independently of other tools. In the third scenario, we assess the effectiveness of *LibRARIAN*’s feature types at identifying versions of native libraries.

Comparative Analysis

OSSPolice uses source code to build an index that allows it to identify versions of binaries. *OSSPolice* measures the similarity between strings extracted from binaries and features found directly in source repositories. Unlike *LibRARIAN*, *OSSPolice* relies on comparing binaries with source code, resulting in an overly large feature space which, in turn, makes *OSSPolice* susceptible to falsely identifying any binary containing a library as exactly matching that library. For example, *OSSPolice* falsely identifies *MuPDF* and *OpenCV* as matching *Libpng* because those two libraries include *Libpng* in their source code [81].

We repeatedly contacted the *OSSPolice* authors to obtain a fully-working version of their tool, but unfortunately they did not provide us their non-public data index or sufficient information to reproduce their setup. As a result, we performed a comparative analysis between *LibRARIAN* and *OSSPolice* based on the published *OSSPolice* numbers [81].

The ground-truth dataset in the *OSSPolice* evaluation contains a total of 475 binaries (out of which 67 are unique) extracted from 104 applications collected by F-Droid [89]. *LibRARIAN* correctly identified 63/67 (94%) unique binaries in the *OSSPolice* dataset, improving accuracy by 12% compared to the accuracy reported by *OSSPolice* (82%) which correctly identified 55/67 libraries. *OSSPolice* has lower accuracy because it misidentifies reused libraries (as described above) and it relies on simple syntactic features (e.g., string literals and exported functions) while our feature vectors extract additional features—such as imported functions, exported and imported global variables, and dependencies that uniquely identify different versions of binaries. These additional features were a major factor in the superior accuracy of *LibRARIAN* compared to *OSSPolice*.

LibRARIAN did not identify 4 binaries because the library functions are dispatched from a single function and do not contain identifying version information that was readily available. Hence, our extracted features fail to provide sufficient information about the underlying

components in the library. Nevertheless, *LibRARIAN* significantly reduces the number of binaries that need to be manually inspected.

Lastly, it is important to reiterate that these results are only compared against the dataset used in the *OSSPolice* paper but without us being able to replicate or reuse *OSSPolice*, due to key elements of the tool being unavailable.

Finding 1: *LibRARIAN* achieves a 12% improvement in its accuracy compared to *OSSPolice* on the 67 unique binaries in *OSSPolice*’s dataset. Unlike *OSSPolice*, *LibRARIAN* obtains this improvement without relying on source code, which may not be available for all libraries and results in an unnecessarily larger feature space.

Independent Accuracy

We further assess *LibRARIAN*’s accuracy on a larger and more recent set of library versions than those found in *OSSPolice*’s dataset. To that end, we manually collect a set of binaries with known libraries and versions (*Known Lib Versions* in Figure 3.1) and compare the inferred libraries and versions to the known ones to determine *LibRARIAN*’s accuracy. We build our dataset based on libraries used in common Android apps.

Experiment Setup. We first manually locate the pre-built binaries of libraries to serve as ground truth. To that end, we use readily available auxiliary data such as keywords found in feature vectors, binary filenames, and dependencies. Once we identify potential targets, we retrieve the pre-built binaries of all versions and architectures, if possible.

There are a variety of distribution channels where app developers can obtain third-party binaries. We obtained such binaries from official websites, GitHub, and Debian repositories. The binaries with known libraries and versions contain 46 distinct libraries with a total of 904 versions and an average of 19 versions per library.

Results. *LibRARIAN* correctly identified the versions of 824/904 (91.15%) libraries in our ground truth: 553/904 (61.17%) of these library versions have unique feature vectors; 15.16% of the these libraries contain the exact version number in the strings literals; and the remaining 14.82% of library versions are distinguished using hash codes to break ties between bin^2sim values of binaries.

Misidentification occurs in 8.85% of library versions, where the largest equivalence class contains 4 library versions. This usually occurs for consecutive versions—minor or micro revisions (e.g., 3.1.0 and 3.1.1). These minor or micro revisions generally fix small bugs and do not change, add, or remove exported symbols. Although *LibRARIAN* cannot pinpoint the exact library version in this case, *LibRARIAN* significantly reduces the search space for post analysis to a few candidate versions.

Finding 2: *LibRARIAN* correctly identifies 824 of 904 (91.15%) library versions from 46 distinct libraries, making it highly accurate for identifying the native libraries and versions. For misidentified library versions, *LibRARIAN* reports a slightly different version.

Feature Effectiveness

To assess the effectiveness of *Metadata Features*, *Version Identification Strings*, and their combination at inferring binaries, we computed the extent to which each feature is capable of inferring binaries in our repository. To that end, any binary whose library and version can be inferred with a bin^2sim above 0.85 as described in Section 3.2 counts as an inferred binary. We found that 37.42% of binaries in our repository are inferable by *Version Identification Strings* only. 45.29% of the remaining binaries are inferable using only the five *Metadata Features* mentioned in Section 3.2.1, while the remaining 17.29% are inferred using both *Metadata Features* and *Version Identification Strings*. This indicates that not all libraries have the version information encoded directly in the strings. Having a combination of both *Metadata Features* and *Version Identification Strings* is crucial to increase the number of

inferred binaries.

Feature Type	Name	Contribution Factor
Metadata	Exported Globals	3.32%
	Imported Globals	1.06%
	Exported Functions	58.25%
	Imported Functions	32.98%
	Dependencies	4.39%

Table 3.3: List of features bin^2sim extracted from native binaries of Android apps along with their type and overall contribution factor, which measures the average percentage each feature contributes to the total similarity score

We further aimed to assess the extent to which each of the five *Metadata Features* contribute to computing bin^2sim in order to assess each of their individual effectiveness. Recall from Section 3.2.2 that our matching algorithm leverages five features when computing the similarity scores between an app binary and our ground-truth dataset. Table 3.3 lists these feature along with their contribution factor, i.e., the average percentage each one of these features contribute to the total similarity score. To calculate the contribution factor ($contrib_f$) of a feature f , we first calculate the similarity score taking all five features into account ($score_{all}$). We then calculate the similarity score of each feature separately ($score_f$). For each feature, we find $contrib_f = score_f / score_{all}$, which is the percentage each f contributes to the total similarity score. As shown in Table 3.3, *Exported Functions* contributes the most when computing bin^2sim (Equation 3.1), i.e., 58.25% of the matching features are *Exported Functions*, followed by *Imported Functions* contributing 32.98%, *Dependencies*, *Exported Globals*, and finally *Imported Globals* contributing less overall. Still, these five features sometimes manage to uniquely identify a library and are therefore included as they, overall, improve the similarity score. Recall that *Version Identification Strings* is not taken into account when computing the similarity score between binaries.

Finding 3: 37.42% of binaries are inferable using *Version Identification Strings*, 45.29% are inferable using *Metadata Features*, and 17.29% are inferable using both feature types. *Exported Functions* and *Imported Functions* account for the overwhelming majority of effectiveness of *Metadata Features*, contributing 58.25% and 32.98%, respectively.

3.3.2 RQ2: Prevalence of Vulnerable Libraries

To study the prevalence of vulnerabilities in native libraries, we need to identify their exact versions. To that end, we leverage *LibRARIAN* to identify potential library versions from our repository. Once the versions are identified, we investigate the extent to which native libraries of Android apps are vulnerable and remain vulnerable.

Experiment Setup. We infer the correct version of 7,253 binaries (10.87% of the total binaries in our Android repository) using *LibRARIAN*. Due to the highly time-consuming nature of the manual collection of ground-truth binaries, we limit ourselves to libraries that (i) are found in a greater number of apps (more than 10 apps) and (ii) have known *CVEs*. As a result, an overwhelming majority of the remaining binaries in our dataset have either no known *CVEs* or affect very few apps, making them an unsuitable choice for applying an expensive manual analysis for studying this research question.

Lib Name	No. Vul Lib Vers	Vul Lib Vers	No. Apps	No. Apps Still Vul
OpenCV	5	2.4.1, 2.4.11, 2.4.13, 3.1.0, 3.4.1	21	7
WebP	3	0.3.1, 0.4.2, 0.4.3	11	1
GIFLib	2	5.1.1, 5.1.4	15	1
FFmpeg	9	2.8, 2.8.7, 3.0.1, 3.0.3, 3.2, 3.3.2, 3.3.4, 3.4, 4.0.2	8	1
Libavcodec	9	55.39.101, 55.52.102, 56.1.100, 56.60.100, 57.107.100, 57.17.100, 57.24.102, 57.64.100, 57.89.100	10	0
Libavformat	3	55.19.104, 56.40.101, 57.71.100	3	0
Libavfilter	3	3.90.100, 4.2.100, 5.1.100	1	0
Libavutil	3	52.48.101, 52.66.100, 54.20.100	2	0
Libswscale	3	2.5.101, 3.0.100, 4.0.100	5	1
Libswresample	2	0.17.104, 1.1.100	1	0
SQLite3	7	3.11.0, 3.15.2, 3.20.1, 3.26.0, 3.27.2, 3.28.0, 3.8.10.2	7	2
XML2	1	2.7.7	3	1
OpenSSL	22	1.0.0a, 1.0.1c, 1.0.1e, 1.0.1h, 1.0.1i, 1.0.1p, 1.0.1s, 1.0.2a, 1.0.2f, 1.0.2g, 1.0.2h, 1.0.2j, 1.0.2k, 1.0.2m, 1.0.2o, 1.0.2p, 1.0.2r, 1.1.0, 1.1.0g, 1.1.0h, 1.1.0i, 1.1.1b	13	3
Jpeg-turbo	2	1.5.1, 1.5.2	3	0
Libpng	7	1.6.10, 1.6.17, 1.6.24, 1.6.34, 1.6.37, 1.6.7, 1.6.8	5	1

Table 3.4: A list of libraries with reported *CVEs* found in our repository along with the number of distinct apps that were affected by a vulnerable library and the number of distinct apps containing a vulnerable version till now.

Results. We found that, out of 7,253 binaries for which we inferred their versions, 3,674 were vulnerable libraries (50.65%) affecting 53/200 distinct apps. 14 new releases of these

distinct apps remain vulnerable at the time of submission. The complete list of libraries with reported *CVEs* between Sept. 2013 and the writing of this chapter can be found in Table 3.4. As for the number of apps affected by vulnerable libraries, our results show that 53 distinct apps have been affected by a minimum of 1 vulnerable library and a maximum of 16 vulnerable libraries covering dates between Sept 2013 and May 2020.

Finding 4: 53 of the 200 top apps on Google Play (26.5%) were plagued by a vulnerable library over approximately six years and 8 months (i.e., between Sept. 2013 and May 2020). 14 of those apps still include a vulnerable binary, i.e., 7% of the top 200 apps on Google Play, even at the time at which we collected apps for this study and are, on average, outdated by 859.17 ± 137.55 days. As a result, vulnerable native libraries play a substantial role in exposing popular Android apps to known vulnerabilities.

We emailed app developers since February 2020 to inform them that their apps continue to use a vulnerable library. We urged them to take an action (i.e., remove or replace such libraries) or at least provide some justification as to why such libraries are not updated. Our investigation is ongoing. While several app developers already updated their apps to remove the vulnerable library, many updates are still outstanding. Some of the replies we received simply blame other library developers. For example, we heard back from Discord that the vulnerable lib is a dependency of another third-party library used in Discord (Fresco): “Until Fresco fixes this, however, we are not able to address this in our app”.

Four libraries were particularly prevalent in terms of the number of vulnerable versions they contain (i.e., *OpenSSL*), the number of apps they affect (i.e., *OpenCV* and *GIFLib*), or the length of time during which the library remained vulnerable (i.e., *XML2* in *Microsoft XBox SmartGlass*). *OpenSSL* has the largest number of vulnerable versions (22 in total) included in 13 distinct apps. 3 apps: Amazon Alexa, Facebook Messenger and Norton Secure VPN still include vulnerable versions of *OpenSSL*.

OpenCV and *GIFLib* affect the most apps. *OpenCV* has the largest number of affected apps with a total of 21 apps where 7 recent apps still have a vulnerable instance of *OpenCV*. Most applications do not include *OpenCV* directly but indirectly through the dependencies of *card.io* which enables card payment processing but comes with the two outdated versions (2.4.11 and 2.4.13) of both *opencv_core* and *opencv_imgproc*. Following *OpenCV* in the number of affected apps is *GIFLib*, which has two vulnerable versions found in a total of 15 distinct apps, 1 app is still affected.

One vulnerable version of *XML2* (2.7.7) was found in 35 versions of *Microsoft XBox SmartGlass* and the library was not updated for 6 years—still remaining vulnerable up to the writing of this chapter. This particular case is notable due to the extremely long amount of time the library had been vulnerable and remained vulnerable.

To examine the affects of vulnerable libraries on apps further, we list popular apps and the reported CVEs they expose their users to. Table 3.5 shows 10 out of 14 popular apps that are using at least one library with a reported CVE at the time of our app collection. We discuss four of these apps in more detail in the remainder of this section.

Facebook Messenger, which has a download base of over 500M (the largest in this list), contains *OpenSSL-1.1.0*, which is vulnerable since Sept. 2016. This vulnerable library contains multiple memory leaks which allows an attacker to cause a denial of service (memory consumption) by sending large OCSP (Online Certificate Status Protocol) request extensions.

Amazon Kindle, an app that provides access to an electronic library of books—with a total of more than 100M installs, uses two vulnerable libraries: *XML2-2.7.7* and *Libpng-1.6.7*. *XML2-2.7.7* contains a variant of the “billion laughs” vulnerability which allows attackers to craft an XML document with a large number of nested entries that results in a denial of service attack. *XML2-2.7.7* is vulnerable since Nov. 2014 and continues to be used in recent versions of the app. *Libpng-1.6.7* has a NULL pointer dereference vulnerability. This

vulnerability was published 6 years ago under *CVE-2013-6954* and it remains unchanged in recent releases of *Amazon Kindle*.

DoorDash, a food delivery app with more than 10M installs includes *GIFLib*-5.1.4 which was reported vulnerable over 8 months ago. A malformed GIF file triggers a division-by-zero exception in the *DGifSlurp* function in *GIFLib* versions prior to 5.1.6. This vulnerable library remains unchanged up to now.

Target uses *OpenCV*-2.4.11 as a dependency of *card.io* which enables card payment processing. This version of *OpenCV* was announced vulnerable in Aug. 2017 yet remains unchanged in these apps.

App Name	Vulnerable Libs	No. Installs
Amazon Alexa	OpenSSL-1.0.2p, SQLite3-3.27.2	10M+
Amazon Kindle	Libpng-1.6.7, XML2-2.7.7	100M+
Amazon Music	FFmpeg-4.0.2	100M+
DoorDash	GIFLib-5.1.4	10M+
Facebook Messenger	OpenSSL-1.1.0	500M+
Grubhub	OpenCV-2.4.1	10M+
Sam's Club	OpenCV-2.4.1	10M+
SUBWAY	OpenCV-2.4.1	5M+
Norton Secure VPN	OpenSSL-1.1.1b	10M+
Target	OpenCV-2.4.11	10M+

Table 3.5: 10 out of 14 popular apps from Google Play which include a vulnerable library that remained unchanged.

Finding 5: These four apps showcase that these vulnerabilities are wide-ranging involving denial of service, memory leaks, or null pointer dereferences. The high severity and long exposure time of these vulnerabilities results in ample opportunity for attackers to target these highly popular apps.

3.3.3 RQ3: Rate of Vulnerable Library Fixing

To determine the vulnerability response rate, we identify the duration between (1) the release time of a security update and (2) the time at which app developers applied a fix either by (i) updating to a new library version or (ii) completely removing a vulnerable library. Recall

that we collected the previous versions of the top 200 apps from Google Play. Moreover, we inferred the library versions from 7,253 libraries using *LibRARIAN*. Given the histories of apps and inferred library versions we can track the library *life span* per app—i.e., the time at which a library is added to an app and when it is either removed or updated to a new version in the app.

To this end, we analyzed 40 popular apps with known vulnerable versions of *FFmpeg*, *GIFLib*, *OpenSSL*, *WebP*, *SQLite3*, *OpenCV*, *Jpeg-turbo*, *Libpng*, and *XML2*, between Sept. 2013 and May 2020. We exclude apps that removed a library before a *CVE* was associated with it and apps containing libraries that are vulnerable up to the writing of this chapter. We obtained the date at which a library vulnerability was found; when a security patch was made available for the library; and the time at which either the library was updated to a new version or removed. Table 3.6 shows all the combinations of apps and vulnerable libraries.

Finding 6: On average, library developers release a security patch after 54.59 ± 8.12 days from a reported *CVE*. App developers apply these patches, on average, after 528.71 ± 40.20 days from the date an update was made available—which is about 10 times slower than the rate at which library developers release security patches.

6 reveals that many popular Android apps expose end-users to long vulnerability periods, especially considering that library developers released fixed versions much sooner. This extreme lag between release of a security patch for a library and the time at which an app developer updates to the patched libraries, or just eliminates the library, indicates that, at best, it is (1) highly challenging for developers to update these kinds of libraries or, less charitably, (2) app developers are highly negligent of such libraries.

Developers applied security patches for vulnerable libraries at a rate as slow as 5.4 years, in the case of *Xbox*, and as fast as 267 days for *Instagram*, where a vulnerable version of *FFmpeg* was removed in that amount of time. In order to determine what type of fix was applied

App Name	Vul Lib Version	Vul Announced	TTRP (Days)	TTAF (Days)
Xbox	XML2-2.7.7	2014-11-04	12	1956
Apple Music	XML2-2.7.7	2014-11-04	12	1704
TikTok	GIFLib-5.1.1	2015-12-21	87	1429
Zoom Meetings	OpenSSL-1.0.0a	2010-08-17	91	1323
Amazon Alexa	OpenSSL-1.0.1s	2016-05-04	12	1086
Amazon Kindle	Libpng-1.6.34	2017-01-30	330	1019
StarMaker	FFmpeg-3.2	2016-12-23	4	1001
eBay	OpenCV-2.4.13	2017-08-06	41	905
Fitbit	SQLite3-3.20.1	2017-10-12	12	902
Uber	OpenCV-2.4.13	2017-08-06	41	830
Snapchat	SQLite3-3.20.1	2017-10-12	12	670
Discord	GIFLib-5.1.1	2015-12-21	87	665
Lyft	OpenCV-2.4.11	2017-08-06	41	662
Twitter	GIFLib-5.1.1	2015-12-21	87	457
Instagram	FFmpeg-2.8.0	2017-01-23	2	267

Table 3.6: Combinations of 15 apps and particular vulnerable library versions they have contained, the date the vulnerability was publicly disclosed (*Vul announced*), the period between vulnerability disclosure and patch availability in days (i.e. Time-to-Release-Patch (*TTRP*)), and the total number of days elapsed before a fix was made (i.e. Time-to-Apply-Fix (*TTAF*))

by a developer, we checked the next app version where a vulnerable library was last seen. We found that developers either kept the library but updated to a new version, removed a vulnerable version, or removed all native libraries in an app. In the next paragraphs, we discuss five popular native libraries used in Android apps that exhibit particularly slow fix rates: *FFmpeg*, *OpenSSL*, *GIFLib*, *OpenCV*, and *SQLite3*.

Multiple vulnerabilities were found in versions 2.8 and 3.2 of *FFmpeg* in Dec. 2016 and Jan. 2017, respectively. The number of days a security patch was released for these vulnerable library versions is 4 and 2 days, respectively. However, developers took 267 days to address vulnerabilities in *Instagram*, and nearly 3 years to apply a fix in *Starmaker*.

OpenSSL-1.0.0a and *OpenSSL-1.0.1s* were associated with *CVE-2010-2939* and *CVE-2016-2105* in Aug. 2010, and May 2016 of which *OpenSSL* developers provided a security patch 91 and 12 days after. However, developers of *Zoom* took 1,323 days to apply a fix, while developers of *Amazon Alexa* took 1,086 days.

A heap-based buffer overflow was reported in *GIFLib*-5.1.1 at the end of 2015. The results show that 3 apps using this vulnerable version of *GIFLib* have an average time-to-fix, i.e., total number of days elapsed before a fix was applied, of 850.33 days (2.3 years), which is 10 times slower. This lag time is particularly concerning since *GIFLib* released a fix 87 days after the vulnerable version.

A fix to an out-of-bounds read error that was affecting *OpenCV* through version 3.3 was released 41 days after the CVE was published. The vulnerable versions of this library affects 3 apps in total: *Uber*, *Lyft*, and *eBay*. *OpenCV* has an average time-to-fix of 799 days (i.e., 2 years), which is 19 times slower than the rate at which library developers of *OpenCV* release security patches.

SQLite3 released version 3.26.0, which fixes an integer overflow found in all versions prior to 3.25.3. *Snapchat* and *Fitbit* removed a vulnerable version of *SQLite-3.20.1* library 786 days later.

Finding 7: The results for these five popular native libraries in Android apps show that it often takes years for app developers to update to new library versions—even if the existing version contains severe security or privacy vulnerabilities—placing millions of users at major risk.

App Name	Time-to-Apply-Fix (Days)	No. Installs
Apple Music	1704.00	50M+
Amazon Kindle	1019.00	100M+
eBay	905.00	100M+
Fitbit	902.00	10M+
Snapchat	844.00	1,000M+
Xbox	763.67	50M+
ZOOM Meetings	668.33	100M+
Lyft	662.00	10M+
Amazon Alexa	605.50	10M+
Uber	588.50	500M+

Table 3.7: Top 10 most negligent apps in terms of the average time to fix a vulnerable library

To further understand the consequences of outdated vulnerable libraries, we calculated

the average time-to-fix across all vulnerable libraries per app. Table 3.7 lists the top 10 apps with the most number of days a vulnerable library remained in an app until a fix for the vulnerability was applied. *Apple Music* had the longest lag between the vulnerable library being introduced and fixed, i.e., 4.66 years. *Uber* was the fastest at almost 589 days. Individual apps had as few as over 10 million installs and as many as over a billion installs. Among the social-media apps, *Snapchat*, which has over 1 billion downloads and the largest number of installs among the top 10 apps in Table 3.7, fixed its vulnerable libraries after 844 days. These very long times to fix vulnerable libraries in highly popular social-media apps places billions of users at high security risk.

Finding 8: The most neglected apps in terms of time to fix vulnerable native libraries range from 588.50 days to nearly five years, affecting billions of users and leaving them at substantial risk of having those libraries exploited. This finding emphasizes the need for future research to provide developers with mechanisms for speeding up this very slow fix rate.

Table 3.8 lists the top 10 most neglected vulnerable libraries across all apps. *XML2* is the most neglected library with an average time-to-fix of 5 years; *WebP* is the least neglected library with an average time-to-fix of 213.40 days. Among these 10 libraries, the fact that it takes app developers 431.81 days, on average, to update vulnerable versions of *OpenSSL* is particularly concerning due to its security-critical nature.

Lib Name	Time-to-Apply-Fix (Days)	Genre
XML2	1830.00	XML parser
Libpng	923.20	Codec
Jpeg-turbo	841.67	Codec
FFmpeg	720.90	Multimedia framework
OpenCV	635.27	Computer Vision
OpenSSL	431.81	Network
GIFLib	421.06	Graphis
SQLite3	369.29	RDBMS
WebP	213.40	Codec

Table 3.8: Top 10 most neglected vulnerable libraries in terms of the average time-to-fix

Finding 9: Future research should focus on these highly neglected libraries as experimental subjects for determining methods to ease the burden of updating these libraries; running regression tests to ensure these updates do not introduce new errors; and repairing those errors, possibly automatically, when they do arise.

3.3.4 Exploitability Case Study

To demonstrate the exploitability of unpatched vulnerabilities in third party apps, we carry out a targeted case study where we analyze individual applications and create a proof-of-concept (PoC) exploit. Our PoC highlights how these unpatched vulnerabilities can be exploited by third parties when interacting with the apps.

XRecorder allows users to capture screen videos, screen shots, and record video calls. Furthermore, *XRecorder* provides video editing functionalities, enabling users to trim videos and change their speed. This application uses FFmpeg, an open-source video encoding framework that provides video and audio editing, format transcoding, video scaling and post-production effects.

XRecorder embeds the FFmpeg library version 3.1.11, which is vulnerable to *CVE-2018-14394* (reported in July 2018). FFmpeg-3.1.11 contains a vulnerable function (*ff_mov_write_packet*) that may result in a division-by-zero error if provided with an empty input packet. Hence, an attacker can craft a WaveForm audio to cause denial of service.

To assess whether this vulnerable function is reachable in *XRecorder*, we used *Radare2* [153] to replace the first instruction in the vulnerable function with an interrupt instruction. We run the application after the latter modification which consequently resulted in an app crash, i.e., allowing us to trigger the vulnerability consistently.

`ff_mov_write_packet` is called by multiple functions across two different binaries (FFmpeg-3.1.11.so and the app-specific libisvideo.so) and two different platforms (Dalvik and Na-

tive). `av_buffersink_get_frame`, one of the ancestors of `ff_mov_write_packet`, is called by `nativeGenerateWaveFormData` from the Dalvik-side.

3.4 Discussion

Findings in *RQ2* (Section 3.3.2) demonstrate that out of 7,253 binaries for which we inferred their versions, 3,674 were vulnerable libraries (50.65%) affecting 53 distinct apps between Sept. 2013 and May 2020. This constitutes about 26.5% of the top 200 apps on Google Play. More alarmingly, new releases of 14 distinct apps remain vulnerable even at the time at which we collected apps for this study with an average outdatedness of 859.17 ± 137.55 days. While we have informed app developers about the outdated libraries in their apps, one interesting piece of follow-up work based on this result is surveying Android app developers to determine the reason for this extremely slow rate of fixing vulnerable native libraries in their apps. Such a study can further assess what forms of support app developers would need to truly reduce this slow rate of updating vulnerable library versions to ones with security patches.

For *RQ3* (Section 3.3.3), we analyzed the speed at which developers updated their apps to patched libraries and found that, on average, library developers release a security patch after 54.59 ± 8.12 days from a reported *CVE*. While app developers apply these patches on average after 528.71 ± 41.20 days from the date an update was made available (10 times slower). Recall that we only consider apps in these cases that actually ended up fixing vulnerable native libraries. The results for *RQ2* and *RQ3* corroborate the need to make app developers aware of the severe risks they are exposing their users to by utilizing vulnerable native libraries.

Overall, our results demonstrate the degree to which native libraries are neglected in terms of leaving them vulnerable. Unfortunately, our findings indicate that the degree of negligence of native libraries is severe, while popular apps on Google Play use native libraries extensively

with 145 out of 200 top free apps (72.50%). Interesting future work for our study includes uncovering the root causes of such negligence and means of aiding developers to quickly update their native libraries. For example, platform providers (e.g., Google) could provide mechanisms to automatically update native libraries while also testing for regressions and possibly automatically repairing them. Such an idea is similar to how Debian’s repositories centrally manage libraries and dependencies between applications and libraries. Whenever a library is updated, only the patched library is updated, the applications remain the same. The Android system would highly profit from a similar approach of central dependency and vulnerability management.

3.5 Threats to Validity

External validity. The primary external threat to validity involves the generalizability of the data set collection and the selection methodology. Recent changes in Google Play limited the length of the “top-apps” list to 200 items. Despite the restrictions imposed by Google Play (limiting our analysis to the top-200 apps), these apps (1) account for the bulk of downloads and the largest user base on Google Play and (2) are generalizable to popular apps, thus having the largest impact.

The results from *RQ1* show that *LibRARIAN* detects versions of native libraries with high accuracy (91.15%). The need to compare against binaries with a known number of versions and libraries (i.e., *Known Lib Versions* in Figure 3.1) limits *LibRARIAN*. Specifically, misidentification of a library or its version might occur when an unknown binary for which we are trying to identify a library and version does not exist in *Known Lib Versions*. In these cases, *LibRARIAN* identifies the unknown binary as being the library and version closest to it according to *bin²sim* that exists in *Known Lib Versions*. One possible way of enhancing *LibRARIAN* in such cases is to leverage supervised machine learning, which may, at least, be able to identify if the library is most likely an unknown major, minor, or patch version of a

known library.

Internal validity. One internal threat is the accuracy of timestamps in AndroZoo and its effect on the reported patch life cycle findings. To mitigate this threat, we collected AndroZoo timestamps over three months and correlated updates with Google Play. We verified that AndroZoo has a maximum lag of 9 days. This short delay is much smaller than the update frequency of vulnerable apps. Furthermore, we verified that using dates added to AndroZoo and version codes give us reliable timestamps for earlier time periods.

Construct validity. One threat to construct validity is the labeling of the libraries in our repository as vulnerable or not. To mitigate this threat, we relied on the vulnerabilities reported by the Common Vulnerabilities and Exposures database [76] which contains a list of publicly known security vulnerabilities along with a description of each vulnerability.

We conducted an exploitability case study of one vulnerable library in an app Section 3.3.4. For the remaining set of discovered vulnerable libraries/apps, we verified that vulnerable native functions are exported and that the library is loaded from the app/Dalvik-side. Performing a complete analysis of exploitable/reachable native functions in Android is an interesting but orthogonal research problem. Building a cross-language control-flow/data-flow analysis to assess reachability of vulnerable native code from the Dalvik code of an Android app is an open research problem, worthy of a separate research paper: (1) recovering a binary CFG/DFG is currently unsound, based on heuristics, and runs into state explosion and (2) conducting an exploitability study of all vulnerable libraries/apps across our entire dataset is infeasible due to the large amount of apps/libraries.

Another threat to validity is the possibility of developers manually patching security vulnerabilities. To mitigate this threat to validity, we checked the versions identified by *LibRARIAN* and found that *LibRARIAN* correctly identifies an overwhelming majority of patch-level versions (61.21%). For the patch-level versions that *LibRARIAN* cannot distinguish as

effectively, *LibRARIAN* makes manual identification much easier, by significantly reducing the search space for post analysis to only 3-4 candidate versions. Furthermore, based on the results of our dataset, we believe that app developers are unlikely to manually patch a library they do not maintain given that it already takes years for these developers to simply update a library version.

3.6 Related Work

A series of work has demonstrated the importance of third-party libraries for managed code of Android apps (i.e., Dalvik code) and their security effects and implications [79, 58]. Derr et al. [79] investigated the outdatedness of libraries in Android apps by conducting a survey with more than 200 app developers. They reported that a substantial number of apps use outdated libraries and that almost 98% of 17K actively used library versions have known security vulnerabilities. Backes et al. [58] report, for managed code-level libraries, that app developers are slow to update to new library versions—discovering that two long-known security vulnerabilities remained present in top apps during the time of their study. None of these studies examined native third-party libraries in Android apps nor did they look at the security impact of vulnerable libraries or whether these vulnerabilities are on the attack surface. *LibRARIAN* now explores the attack surface of native libraries, closing this important gap and calling platform providers to action.

A wide variety of approaches have emerged that identify third-party libraries with a focus on managed code. These approaches employ different mechanisms to detect third-party libraries within code including white-listing package names [103, 64]; supervised machine learning [147, 137]; and code clustering [175, 140, 132]. LibScout [58] proposed a different technique to detect libraries using normalized classes as a feature that provides obfuscation resiliency.

Some techniques identify vulnerabilities in native libraries by computing a similarity score between binaries with known vulnerabilities and target binaries of interest [98][85]. VulSeeker

[98] matches binaries with known vulnerabilities using control-flow graphs and machine learning. Similarly, *discovRE* [85] and *BinXray* [178] matches binaries at the function level. Other techniques employ a hybrid technique such as *BinSim*[143], *Mobilefinder*[133], *BinMatch*[111], and *DroidNative* [49]. These approaches identify semantic similarities/differences between functions in binaries based on execution traces for the purpose of analyzing/identifying malware. Unlike these tools, *LibRARIAN* focuses on benign libraries with the goal of identifying their versions with high scalability.

Binary Analysis Tool (BAT) [108] and *OSSPolice* [81] measure similarity between strings extracted from binaries and features found directly in source repositories. Unlike *LibRARIAN*, these approaches compare source code with binaries, which introduces the issue of internal clones (i.e., third-party library source code that is reused in the source code of another library). BAT and *OSSPolice* rely on simple syntactic features (e.g., string literals and exported functions). *OSSPolice* cannot detect internal code clones, while *LibRARIAN* can, giving it superior ability to identify versions of native libraries. Furthermore, BAT does not detect versions of binaries and was shown to have inferior accuracy for computing binary similarity compared to *OSSPolice*. Unlike these tools, *LibRARIAN* extracts additional features—such as imported functions, exported and imported global variables, and dependencies that uniquely identify different versions of binaries. As shown in Section 3.3.1, these additional features were a major factor in the superior accuracy of *LibRARIAN* compared to *OSSPolice*.

Other related empirical research studies the prevalence of vulnerable dependencies in open source projects [69], vulnerabilities in WebAssembly binaries [127], or investigates the updatability of ad libraries in Android Apps [145]. Other work such as [107, 179] present third party library recommendation tools for mobile apps.

Despite the existence of much previous work on survivability of vulnerabilities in Android apps/libraries, such work has not conducted a large-scale longitudinal study of native third-party libraries as we did in this work. Moreover, the survivability of vulnerabilities in

non-native libraries are significantly shorter compared to those reported in our results. While survivability of vulnerabilities in native Android apps took, on average, 528.71 ± 40.20 days in our study, prior work [53, 141] shows that survival times of vulnerabilities in Python and Javascript are 100 days and 365 days, respectively. 50% of vulnerabilities in npm packages were fixed within a month, 75% were fixed within 6 months only [78].

None of this aforementioned related work has examined the prevalence of security vulnerabilities in Android’s native libraries or the time-to-fix for vulnerable versions of such libraries. As a result, our work covers a critical attack vector that has been ignored in existing research.

Chapter 4

A Comprehensive Study of Autonomous Vehicle Bugs

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 chapter 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.

4.1 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 [38, 39, 18]. In total, there are already about 50 corporations actively developing AVs [9, 2]. 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 [8, 26]. AVs with lower levels of autonomy have resulted in another set of fatalities during recent years [31, 32, 30, 25, 29, 28]. 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 [80], machine learning libraries [113, 181, 167], concurrency bugs [126, 138], performance bugs [114, 159], and error-handling bugs [169, 73, 66]. None of these studies have focused on bugs in AV software systems.

This chapter 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 [15] and Autoware [13]. 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 [34], can be directly deployed on real-world AVs such as Lincoln MKZ [15], and has already reached mass production agreements with Volvo and Ford [17]. For Autoware, it is an open-source system for AVs run by the Autoware Foundation [13], 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 [10, 7].

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 chapter 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 [6].

The rest of the chapter is organized as follows. The next section provides further background on AV software systems. Section 4.3 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 4.4), follow that with a discussion drawing broader lessons from those results (Section 4.5), and detail threats to validity (Section 4.6). Finally, we describe related work (Section 4.7) and conclude.

4.2 Autonomous Vehicle Systems

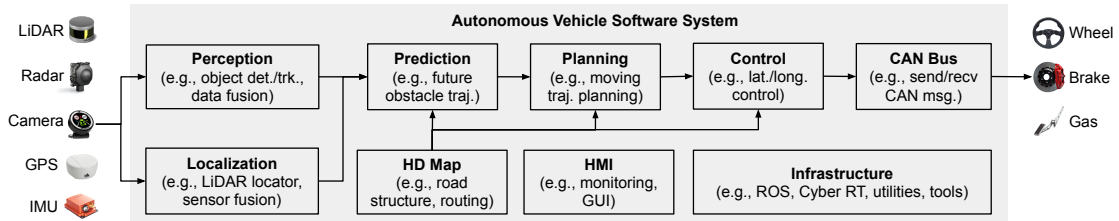


Figure 4.1: State-of-the-art Autonomous Vehicle (AV) software system architecture from most popular AV development classes such as Udacity Self-Driving Car Engineer classes [34] and real-world AV systems such as Baidu Apollo [15] and Autoware [13].

The Society of Automotive Engineers (SAE) defines 6 levels of vehicle autonomy [157], 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. [2], and some of them are already available to

the general public, e.g., the Google Waymo One self-driving taxi service [11]. 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. 5.1 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 [34] and used in representative real-world AV systems such as Baidu Apollo [15] and Autoware [13]. 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 [62] and ICP [109]) 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 [63]) 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 [99] and PID [56] 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 [149]. It also includes a *robotics middleware* (e.g., ROS [152], Cyber RT [15]), 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.

4.3 Methodology and Classification

4.3.1 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 4.1. 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 work 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 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 [171, 80, 113, 181] 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.

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

¹SLOC: source lines of code

Table 4.1: Statistics of Apollo and Autoware from GitHub

4.3.2 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 work. 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 [158, 167, 185, 174]. In this work, we initially adopted the taxonomy of root causes presented in [167, 158] 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 [80, 113, 181, 167]. 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.

4.3.3 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.

4.3.4 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.

4.3.5 Affected AV Components

After the aforementioned labeling process, the following AV components (described in Section 4.2) 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 5.1 and share the same reference architecture. Table 4.2 shows other core components found to have a significant number of bugs after the labeling process was completed.

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.

Table 4.2: Additional Core Components with Significant Bugs

Perception, Localization, and CAN Bus components had major sub-components with significant amounts of bugs. Table 4.3 depicts those sub-components.

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

Table 4.3: AV Sub-Components with Significant Bugs

4.3.6 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 [74], 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?*

4.4 Experimental Results

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

4.4.1 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 4.4. 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 [177, 101, 146].

Root Cause	Apollo	Autoware	<i>Total_{cause}</i>
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
<i>Total_{system}</i>	243	256	499

Table 4.4: Root Causes of Bugs in AV Systems

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 [146].

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

4.4.2 RQ2: AV Bug Symptoms

The next research question we discuss covers the symptoms that AV bugs exhibit. Table 4.5 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.

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

¹<https://tinyurl.com/y5vd6mqu>

Symptom	Apollo	Autoware	<i>Total_{symp}</i>
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
<i>Total_{system}</i>	243	256	499

Table 4.5: Symptoms of Bugs in AV Systems

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 Apollo and Autoware. In particular, Autoware uses the native ROS build system [152], which reuses the *CMake* syntax [5] when specifying the compilation configurations. Apollo, on the other hand, adopts the newer *Bazel* build system [4] developed by Google. However, given that both Autoware and Apollo are built on top of robotics middleware (e.g., ROS [152])

or Cyber RT [15]), 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.

4.4.3 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.

Symptom RootCause	Crash	Hang	Build	Cam	LPN	SVC	Lau	TLP	Stop	Turn	Traj	IO	Loc	OP	Doc	DGUI	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

Table 4.6: Frequency of symptoms that each root cause of a bug may exhibit across Apollo and Autoware.

Table 4.6 illustrates the extent to which a particular root cause resulted in a specific symptom across both AV systems. Recall that 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 [177, 101, 146].

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 4.4.1. 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%).

4.4.4 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 4.2. Table 4.7 presents the number of occurrences for each top-level component, as described in Sections 4.2 and 4.3—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.

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

Component	Sub-Component	Apollo	Autoware	<i>Total_{comp}</i>
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
<i>Total_{system}</i>		243	256	499

Table 4.7: Frequency of bug occurrences for each AV component

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.

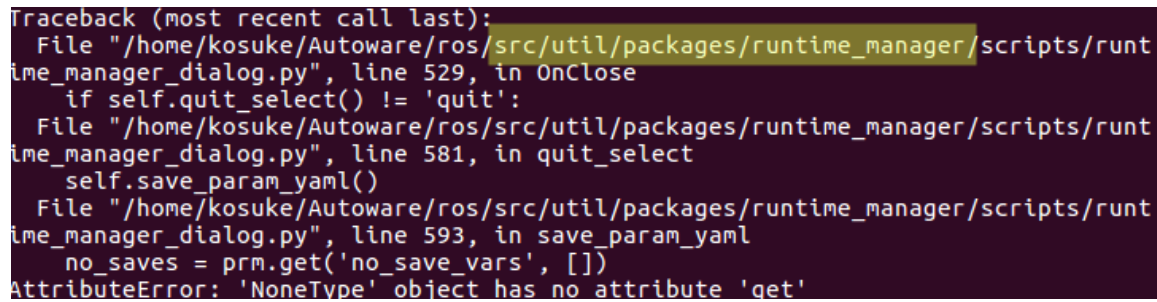
Finding 11: The core AV components with the greatest number of bugs across both systems are Planning, Perception, and Localization—ordered from most bug-ridden to least—with 135 (27.05%), 83 (16.63%), and 57 (11.42%) bugs, respectively.

A substantial number of bugs involve functionality that does not provide core logic that fits

into the major components as described in Section 4.2. 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 4.2 was obtained from one of Autoware’s issues² 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.



```
Traceback (most recent call last):
  File "/home/kosuke/Autoware/ros/src/util/packages/runtime_manager/scripts/runtime_manager_dialog.py", line 529, in onClose
    if self.quit_select() != 'quit':
  File "/home/kosuke/Autoware/ros/src/util/packages/runtime_manager/scripts/runtime_manager_dialog.py", line 581, in quit_select
    self.save_param_yaml()
  File "/home/kosuke/Autoware/ros/src/util/packages/runtime_manager/scripts/runtime_manager_dialog.py", line 593, in save_param_yaml
    no_saves = prm.get('no_save_vars', [])
AttributeError: 'NoneType' object has no attribute 'get'
```

Figure 4.2: A bug found in one of Autoware’s utilities.

4.4.5 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.

²<https://tinyurl.com/yxskk46n>

Component	Symptom	Crash	Hang	Build	Cam	LPN	SVC	Lau	TFP	Stop	Turn	Traj	IO	Loc	OP	Doc	DGUI	SS	Logic	UN	OT
	Sub-Component																				
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	1	2	1	1	1
	Documentation & Others	2	0	16	0	0	1	1	0	1	0	0	0	0	0	38	4	0	1	1	2

Table 4.8: Occurrences of bug symptoms in components of Apollo and Autoware

Table 4.8 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() <
    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

³<https://tinyurl.com/y4v4l7mc>

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%) .

4.5 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 [168, 180]. 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.

4.6 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 [167, 158], 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 [171, 80, 113, 181] 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 [9, 2, 38, 39, 18, 17, 10, 7]. 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 [113] and 175 [181] for deep learning and 269

for numerical libraries [80]). 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 [3, 13]. 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.

4.7 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.* [80] studied the bugs that occur in numerical software libraries such as NumPy, SciPy, and LAPACK. Islam *et al.* [113], Thung *et al.* [167], and Zhang *et al.* [181] investigated machine learning and deep learning frameworks (e.g., Caffe, Tensorflow, OpenNLP, etc.). Leesatapornwongsa *et al.* [126] and Lu *et al.* [138] analyzed concurrency bugs in distributed systems such as Hadoop and HBase. Jin *et al.* [114] and Selakovic *et al.* [159] 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 [80, 113, 167, 181, 126, 138, 114, 159], 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 [138]. 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 [31, 32, 30, 25, 29, 28]. Uber’s AV system reportedly failed to prevent the crash after detecting the pedestrian 6 seconds before the accident in Tempe, AZ [8, 26]. Moreover, machine learning models used in AV systems (e.g., in Perception) have been found vulnerable to attacks (e.g., physical-world perturbations [88, 87, 183] or sensor attacks [68]). 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.

Chapter 5

Generating Diverse, Fully-Mutable, Safety-Critical and Motion Sickness-Inducing Scenarios for Autonomous Vehicles

Autonomous Vehicles (AVs) leverage advanced sensing and networking technologies (e.g., camera, LiDAR, RADAR, GPS, DSRC, 5G, etc.) to enable safe and efficient driving without human drivers. Although still in its infancy, AV technology is becoming increasingly common and could radically transform our transportation system and by extension, our economy and society. As a result, there is tremendous global enthusiasm for research, development, and deployment of AVs, e.g., self-driving taxis and trucks from Waymo and Baidu. The current practice for testing AVs uses virtual tests—where AVs are tested in software simulations—since they offer a more efficient and safer alternative compared to field operational tests. Specifically, search-based approaches are used to find particularly critical situations. These approaches provide an opportunity to automatically generate tests; however, systematically

creating *valid* and *effective* tests for AV software remains a major challenge. To address this challenge, we introduce SCENORITA, a test generation approach for AVs that uses evolutionary algorithms with (1) a novel gene representation that allows obstacles to be *fully mutable*, hence, resulting in more reported violations, (2) 5 test oracles to determine both safety and motion sickness-inducing violations, and (3) a novel technique to identify and eliminate duplicate tests. Our extensive evaluation shows that SCENORITA can produce effective driving scenarios that expose an ego car to safety critical situations. SCENORITA generated tests that resulted in a total of 1,026 *unique* violations, increasing the number of reported violations by 23.47% and 24.21% compared to random test generation and state-of-the-art partially-mutable test generation, respectively.

5.1 Introduction

Autonomous vehicles (AVs), a.k.a. self-driving cars, are becoming a pervasive and ubiquitous part of our daily life. More than 50 corporations are actively working on AVs, including large companies such as Google’s parent company Alphabet, Ford, and Intel [36, 20, 37]. Some of these companies (e.g., Alphabet’s Waymo, Lyft, and Baidu) are already serving customers on public roads [38, 39, 18]. Experts forecast that AVs will drastically impact society, particularly by reducing accidents [61]. However, crashes caused by AVs indicate that achieving this lofty goal remains an open challenge. Despite the fact that companies such as Tesla [27], Waymo [36], or Uber [33] have released prototypes of AVs with a high level of autonomy, they have caused injuries or even fatal accidents to pedestrians. For instance, an AV of Uber killed a pedestrian in Arizona back in 2018 [26]. AVs with lower levels of autonomy have resulted in more fatalities in recent years [31, 32, 30, 25, 29, 21, 28, 26].

Prior research has revealed a lack of standardized procedures to test AVs [120] and the inability of current approaches to effectively translate traditional software testing approaches into the space of AVs [124, 112]. A common practice for testing AV software lies in field

operational tests, in which AVs are left to drive freely in the physical world. This approach is not only expensive and dangerous, but also ineffective since it misses critical testing scenarios [116]. Virtual tests, where AVs are tested in software simulations, offer a far more efficient and safer alternative. While these tests provide an opportunity to automatically generate tests, they come with the key challenge of *systematically generating scenarios which expose AVs to safety-critical and motion sickness-inducing situations*.

To address this challenge, we propose SCENORITA (**s**cenario **Gene**Rat**Ion** **T**esting for **AVs**), a test generation framework which aims to find safety and motion sickness-inducing violations in the presence of an evolving traffic environment. SCENORITA combines both (i) AV software domain knowledge and (ii) search-based testing [142, 95]. These two elements have been combined by previous techniques to test AVs by automatically generating safety-critical scenarios [130, 97, 67, 52, 60, 47, 184]. However, unlike these approaches, SCENORITA’s gene representation enables obstacles to be *fully mutable*, i.e., an obstacle’s *individual* properties such as its start and end location, type (e.g., vehicle, pedestrian, and bike), heading, speed, size, and mobility (e.g., static or dynamic) can be altered. Previous techniques do not specify their gene representations or do so in such a way that allows obstacles to be only *partially mutable*: obstacles’ attributes are altered only during mutation and with a small probability, while during crossover, obstacles are transferred across scenarios without altering their states or properties [130, 96, 97, 67, 184]. Thus, these techniques ignore the challenge of ensuring creation of valid obstacle trajectories, reducing their *effectiveness at generating driving scenarios with unique violations*.

Other limitations of prior work include generating driving scenarios with: (i) manually setup and limited number of scenario types; (ii) a small number of obstacles per scenario (a maximum of 2 obstacles per scenario); and (iii) obstacles with fixed trajectories and limited maneuvers which require manual specification. SCENORITA’s gene representation allows obstacles to fully evolve throughout the test generation process, while still adhering to traffic

laws and providing the ego car with ample amount of time to react to any potential violations. This novel gene representation addresses the limitations in the state-of-the-art approaches (Section 5.2), as follows:

- As shown later in our evaluation results (Section 5.5), a fully-mutable version of SCENORITA produces 23% more violations compared to a partially-mutable version of SCENORITA. Both representations contain the same components described in Figure 5.1, they only differ in the gene representation.
- SCENORITA eliminates the need for manual and fixed scenario setup by doing the following: (i) given any 3D map, SCENORITA parses the map and generates a directed graph of all points (nodes) residing in the map and their connected lanes (edges), (ii) this directed graph is used to validate the trajectories of ego car and obstacles at the start of the test generation, and as they evolve and mutate during the evolution process. In other words, there is no need to manually set a fixed trajectory for the ego car or obstacles, as SCENORITA handles that automatically.
- Since there is no need to manually setup any scenarios (i.e., obstacles and ego car can be placed anywhere in the map and it is ensured that they adhere to traffic laws and are within acceptable proximity), this allows SCENORITA to generate a much larger, diverse and complex set of scenario types which: (i) cover as many lanes and lane types, in a map, as possible (ex. single- or multi-lane roads with either same or opposite traffic direction, U-turns, roundabouts, cross or T intersections, merged lanes, etc.); (ii) does not have a limited or fixed number of obstacles; (iii) supports any combination of maneuvers per scenario for both an ego car and an obstacle. For example, in one scenario, an ego car can be directed to follow an obstacle before it changes lanes and then turns right at an intersection.

Additionally, previous work on AV software testing uses a highly limited number of test oracles for ensuring safety and no oracles for assessing motion sickness-inducing movement of

an AV: State-of-the-art AV testing approaches (*AC3R*, *AV-Fuzzer*, *AsFault*, *AutoFuzz* and Abdessalem et al. [60, 47]) use only two oracles for checking if (1) the ego car reaches its final expected position while avoiding a crash (i.e., collision detection) and (2) if a vehicle drives off the road (i.e., off-road detection). As a result, the fitness functions these techniques utilize are overly simplified—substantially reducing their degree of safety assurance while completely ignoring rider comfort and motion sickness. Research has shown that a rider’s discomfort increases when a human is a passenger rather than a driver—with up to one-third of Americans experiencing motion sickness, according to the National Institutes of Health (NIH) [14, 23, 128].

To overcome such limitations, SCENORITA utilizes 5 test oracles (i.e., collision detection, speeding detection, unsafe lane change, fast acceleration, and hard braking) and corresponding fitness functions—which are based on grading metrics for driving behavior defined by Apollo’s developers [19]. Apollo is a high autonomy (i.e., Level 4), open-source, production-grade AV software system created by Baidu. The Society of Automotive Engineers (SAE) defines 6 levels of vehicle autonomy [157, 57], where Level 4 (L4) AV systems, such as Apollo, have the AV perform all driving functionality under certain circumstances, although human override is still an option. Apollo is selected by Udacity to teach state-of-the-art AV technology [34] and can be directly deployed on real-world AVs such as Lincoln MKZ, Lexus RX 450h, GAC GE3, and others [15, 24], and has mass production agreements with Volvo and Ford [17]. Additionally, Apollo has already started serving the general public in cities (e.g., a robo-taxi service in Changsha, China [16]).

The main contributions of this chapter are as follows:

- We introduce SCENORITA, a search-based testing framework, with a novel gene representation and *domain-specific constraints*, that automatically generates *valid* and *effective* driving scenarios. SCENORITA aims to maximize the number of scenarios with unique violations and relies on a novel gene representation of driving scenarios, which enables the

search to be more *effective*: Our gene representation allows the genetic algorithm to alter the states and properties of obstacles in a scenario, allowing them to be fully mutable. Additionally, we specify a set of *domain-specific constraints* to ensure that the generated driving scenarios are *valid*. To the best of our knowledge, we are the first to define the exact values of these constraints, which are obtained from authoritative sources such as the National Center for Health Statistics, Federal Highway Administration, and the US Department of Transportation [71, 91, 90, 92].

- To improve the effectiveness of SCENORITA, we automate the process of identifying and eliminating duplicate violations by using an unsupervised clustering technique to group driving scenarios, with similar violations, according to specific features.
- We utilize 5 test oracles and corresponding fitness functions to assess different aspects of AVs—ranging from traffic and road safety (i.e., collision detection, speeding detection, and unsafe lane change) to a rider’s comfort (i.e., fast acceleration and hard braking). To the best of our knowledge, SCENORITA is the first search-based testing technique for AV software that uses multiple test oracles at the same time and considers both comfort and safety violations as part of those oracles.
- We evaluate the efficiency and effectiveness of SCENORITA by comparing it with random search and a state-of-the art search-based approach—which adopts the gene representation in prior work [130, 67, 97, 184] that allows *only* partial mutation of obstacles (i.e., obstacles are transferred across scenarios without altering their states or properties during crossover).

Our extensive evaluation—which consists of executing a total of 31,413 virtual tests on Baidu Apollo using 3 high-definition maps of cities/street blocks located in California: Sunnyvale (3,061 lanes); San Mateo (1,305 lanes); and Borregas Ave (60 lanes)—shows that SCENORITA generates driving scenarios that expose the ego car to critical and realistic situations. SCENORITA found 1,026 *unique* comfort and safety violations, while random testing and the partially mutable search-based testing found a total of 826 and 831, respectively. We make our testing platform, dataset and results available online to enable reusability, reproducibility,

and others to build upon our work [1].

Table 5.1: Comparing SCENORITA with the related work.

	scenoRITA	AV-Fuzzer [130]	AutoFuzz[184]	Avoidable collisions[67]	AsFault[97]
Main Objective	Generate driving scenarios that expose the ego car to 3 types of safety-critical and 2 types of motion sickness-inducing scenarios	Find collision violations of autonomous vehicles in the presence of an evolving traffic environment	A grammar-based fuzzing technique for finding collision and off-road violations in AV controllers	A search-based approach to find avoidable collisions	Automatically create virtual roads for testing lane-keeping of self-driving car software
Requires High Definition (HD) Maps	Yes	Yes	Yes	Yes	No
Representations (gene)	Fully Mutable	Partially Mutable	Not Known	Partially Mutable	Virtual Roads only (no obstacles included)
Scenario Types	Unlimited	Limited (5 scenarios)	Limited (5 scenarios)	Limited (7 scenarios)	N/A
Scenario Configuration	Automated	Manual	Manual	Manual	N/A
Supported Maneuvers	Supports any combination of the following maneuvers: Lane Follow, Lane Change, Left Turn, Right Turn, Cross Intersection, U-Turn, Lane Merge, Acceleration, Deceleration.	Supports one maneuver at time from 4 possible maneuvers: Acceleration, Deceleration, Lane Follow, and Lane Change.	Supports one maneuver at time from 6 possible maneuvers: Acceleration, Deceleration, Turn Left, Turn Right, Lane Follow, Lane Change.	Not Known	N/A
Ego Car Routing	Flexible	Not Known	Fixed	Not Known	Flexible
No. Obstacles Per Scenario (for experiments)	Up to 70	Maximum of 2	Maximum of 2	No Known	No obstacles
Collision Detection	Supported	Supported	Supported	Supported	Not Supported
Speeding Detection	Supported	Not Supported	Not Supported	Not Supported	Not Supported
Unsafe Lane Change	Supported	Not Supported	Supported (only in Carla, not Apollo)	Not Supported	Supported
Fast Acceleration	Supported	Not Supported	Not Supported	Not Supported	Not Supported
Hard Braking	Supported	Not Supported	Not Supported	Not Supported	Not Supported

5.2 Related Work

A wide array of studies focus on **applying traditional testing techniques to AVs** including adaptive stress testing [75], where noise is injected into the input sensors of an

AV to cause accidents; fitness function templates for testing automated and autonomous driving systems with heuristic search [106]; and search-based optimization [122]. These studies provide limited insights into the testing of real-world AVs, since they do not evaluate their techniques on open-source, production-grade AV software.

Other related work focuses on the **vision and machine-learning aspects of AV** software [186, 165, 180, 121, 151, 47, 60, 104]. Rather than focus on these aspects, SCENORITA targets the planning component of AV software. Previous work has shown that the most bug-ridden component of production-grade, open-source AV software systems is the planning component as opposed to the components responsible for or utilizing vision or machine-learning capabilities of AVs [100].

Reproducing tests from real crashes: crashes are recreated by replaying the sensory data collected during physical-world crashes in [84]. Similarly, *AC3R* [96] generates driving simulations which reproduces car crashes from police reports using natural language processing (NLP). However, *AC3R* requires manual collection of police reports and inherits the accuracy limitations of the underlying NLP used to extract information from police reports.

Search-based procedural road generation: *AsFault*[97] uses procedural content generation and search-based testing to automatically create challenging virtual scenarios for AV software. Similarly, tools published as part of the Search-Based Software Testing Challenge (SBST) [44, 45] generate challenging road networks for virtual testing of an automated lane keep system such as GABezier [123], Frenetic [70], and Deeper [144]. However, none of these tools take into account the behaviour of other obstacles when testing for safety violations in AVs. Other tools in the SBST Challenge derive tests for Java such as EvoSuite [155] and Kex [48]. None of the latter tools are targeted to generate tests for autonomous vehicles.

Next we discuss **interesting but orthogonal research problems** such as Li et. al. [131] which discussed the original idea to consider safety and comfort of autonomous vehicles.

However, this work does not formalize or encode safety and comfort—when evolving driving scenarios—to produce tests that expose the autonomous vehicle to safety and comfort violations. This work, instead, aims to provide a quantitative way to measure safety and comfort of autonomous driving in a test. Another major distinction between scenoRITA and Li et. al.’s approach is that the latter is not a fully automatic test generation framework. This approach requires a human expert to vaguely define “test tasks” and perform qualitative judgments before the simulation-based system can make more precise task definitions and generate more tests. The human expert then provides feedback to the simulation system to validate test results.

Another orthogonal related work is by Calò et al. [67] which proposed two search-based approaches for finding *avoidable* collisions. They define comfort and speed as weights to rank short-term paths; however, they do not formalize comfort and speed in the fitness function, nor do they evaluate them. The main focus of our approach is not introducing a sophisticated approach to evaluate violations’ “avoidability” but to find as many violations as possible. Nonetheless, we still ensure that the generated tests are valid using simplistic time-, speed- and location-related thresholds to determine “avoidable” violations.

As opposed to finding safety and comfort violations, Luo et. al. propose a framework (EMOOD) [139] that evolves tests to identify combinations of requirements violations. In other words, EMOOD aims to find different requirements violation patterns for two reasons: (i) different combinations of requirements violations can expose different types of failures. For example, the type of failure in which the autonomous vehicle collides while running a red light is different from the one in which the autonomous vehicle collides while violating the lane keeping, as the different combinations of requirements violations may provide different insights about the cause of the collisions; (ii) satisfying all requirements may not be possible for an ADS in practice, as unexpected events may happen in highly open and dynamic environments. In response to these unexpected events, the control software of the ADS has

to make trade-offs among requirements, likely resulting in one or more requirements being violated.

Table 5.1 compares SCENORITA with the state-of-the-art techniques; AsFault[97], and Avoidable Collisions [67] address interesting but orthogonal problems to our approach, AV-Fuzzer and AutoFuzz are closely related to SCENORITA. *Scenario Types* in Table 5.1 refer to the number of different scenarios supported by each approach. A scenario type is determined by (i) a specific part of a map (ex. intersection X in map Y), (ii) the number of obstacles in that scenario, (iii) the movement path/routing of obstacles and the ego car (ex. obstacle O starts at point A and ends at point B), (iv) the allowed maneuvers of obstacles and the ego car (ex. obstacle O only turns right at an intersection). For example, [82] generates tests for **one scenario type** only, which consists of **one pedestrian** crossing the **same street** in a given map, with fixed trajectories for both the ego car and pedestrian, throughout the test generation process. The approaches depicted in Table 5.1 and similarly [41, 82, 161, 119, 148, 172, 139] fail to find diverse/unique violations due to the limited number and lack of diversity in generated scenarios (the maximum number of scenario types supported by any of these tools is 7 scenario types).

Scenario Configuration in Table 5.1 refers to the means used to set up a *Scenario Type*. For example, to set up a *Scenario Type* similar to Figure 5.6(a) in *AutoFuzz* [184], one must manually set (i) start and end location of the ego car in a specific map, (ii) allowed maneuver of the ego car (ex. in Figure 5.6(a), the ego car always turns left), (iii) the number and type of obstacles (ex. in Figure 5.6(a), there’s one car and one pedestrian), (iv) start and end locations for every possible obstacle in that scenario (ex. in Figure 5.6(a), one must specify the routing info of the blue car and the pedestrian). Any slight difference in one scenario type in *AutoFuzz*, requires manually modifying its configuration. We believe that the reason for the limited number of scenario types and limited maneuvers in related work is due to the fact that these tools require a manual setup for driving scenarios. In other words, if we would

want to generate a scenario other than the ones specified by the authors of these tools, we would have to decide what part of a specific map we would want to run our tests on, what number of obstacles we would want to include, what is the movement path for each one of these obstacles, and where would we want to place the ego car in the map. Once we set up a scenario, all the generated tests will be executed for that specific scenario only.

Furthermore, the need to manually setup these scenarios can be attributed to the gene representation used in related work. These tools avoid applying search operators on an obstacle attribute-level (unless its speed) due to the tools’ inability to generate obstacles with valid new trajectories. For example, applying a crossover between two obstacles can result in an offspring with a new movement path (trajectory), hence, there needs to be a mechanism which ensures that the newly generated obstacle has a valid trajectory (i.e., travels in the direction of traffic, with close proximity to the ego car but still allows the ego car ample amount of time to react to any potential violations, not placed in the middle of intersections, etc). The challenging aspect in generating complex and diverse scenarios is to be able to evolve obstacles fully regardless of the initial setup of such obstacles in a scenario. Related work, evolve obstacles in tests by either: (i) transferring obstacles across scenarios without altering their states or properties, or (ii) mutating the speed of such obstacles or moving them to neighboring lanes. As a result, these works opt for using a fixed, and manually setup scenarios—throughout test generation—with obstacles and the ego car having fixed trajectories and limited maneuvers.

Ego Car Routing in Table 5.1 indicates whether an approach supports *flexible* ego car routing (i.e., an ego car can traverse different paths in a map, covering as many roads of the map as possible) or *fixed* routing (i.e., ego car traverses the same path over and over again), hence, must be defined manually.

Moreover, the approaches described in this section, only find collision violations [41, 82, 161, 119, 148, 130, 184, 67] or lane keeping [172, 97, 123, 70, 144]. None of them evolve tests for

a combined set of safety violations (collision, speed, and unsafe lane change) and none of them report comfort violations.

5.3 Specification of the State Space

To aid in the generation of effective and valid scenarios, we present a formal specification of the state space in the form of driving scenarios. SCENORITA uses this formal specification of the state space, along with a genetic algorithm, to generate scenarios that maximizes the possibility of the ego car (i.e., AV) either violating safety or causing rider discomfort.

Definition 1. A Scenario \mathcal{S}_c is a tuple $\langle t, E, \mathbb{O}, \mathbb{L} \rangle$ where:

- t is a finite number that represents the maximum duration of \mathcal{S}_c .
- E is the only ego car (i.e., the autonomous driving car) in \mathcal{S}_c .
- \mathbb{O} is a finite, non-empty set of n obstacles (i.e. non-player characters). A *single* obstacle is represented as O_k where $\mathbb{O} = \{O_k : 1 \leq k \leq n\}$.
- \mathbb{L} is a finite, non-empty set of lanes, where E and \mathbb{O} reside/travel.

Definition 2. An ego car E , is a tuple

$\langle Z_E, \mathbb{H}_E, \mathbb{P}_E, \mathbb{S}_E, \mathbb{A}_E, \mathbb{C}_E \rangle$ where:

- $Z_E = \langle wid, len, hgt \rangle$ represents the width wid , length len , and height hgt of the ego car E .
- \mathbb{H}_E is a finite, non-empty set representing the ego car's headings during time instants of \mathcal{S}_c . The heading of E at timestamp j is represented as h_j^E where $\mathbb{H}_E = \{h_j^E : 1 \leq j \leq t\}$.
- \mathbb{P}_E is a finite, non-empty set representing the ego car's positions during time instants of \mathcal{S}_c . The position of E at timestamp j is represented as p_j^E where $\mathbb{P}_E = \{p_j^E : 1 \leq j \leq t\}$.
- \mathbb{S}_E is a finite, non-empty set representing the ego car's speed during time instants of \mathcal{S}_c . The speed of E at timestamp j is represented as s_j^E where $\mathbb{S}_E = \{s_j^E : 1 \leq j \leq t\}$.
- \mathbb{A}_E is a finite, non-empty set representing the ego car's acceleration at time instants of \mathcal{S}_c . The acceleration of E at timestamp j is represented as a_j^E where $\mathbb{A}_E = \{a_j^E : 1 \leq j \leq t\}$.

- \mathbb{C}_E is a finite, non-empty set of durations an ego car spends driving at the boundary of two lanes at the same time. When an ego car changes lanes, it drives on the markings between two lanes for a period of time c , before it completely switches to the target lane. The duration E spends driving on the markings at timestamp j is represented as c_j^E where $\mathbb{C}_E = \{c_j^E : 1 \leq j \leq t\}$.

Definition 3. A *single* obstacle O_k in \mathcal{S}_c is a tuple

$\langle ID_{O_k}, T_{O_k}, Z_{O_k}, M_{O_k}, \mathbb{H}_{O_k}, \mathbb{P}_{O_k}, \mathbb{S}_{O_k} \rangle$ where:

- ID_{O_k} represents a unique identification number associated with O_k .
- T_{O_k} represents the type of an obstacle. Examples of obstacle types are: VEHICLE, PEDESTRIAN, and BICYCLE.
- $Z_{O_k} = \langle wid, len, hgt \rangle$ represents the width wid , length len , and height hgt of obstacle O_k .
- M_{O_k} represents the mobility of an obstacle (e.g., static or mobile).
- \mathbb{H}_{O_k} is a finite, non-empty set representing the direction of O_k during the entire duration of \mathcal{S}_c . The heading of O_k at timestamp j is represented as $h_j^{O_k}$ where $\mathbb{H}_{O_k} = \{h_j^{O_k} : 1 \leq j \leq t\}$.
- \mathbb{P}_{O_k} is a finite, non-empty set representing the positions of O_k at time instants of \mathcal{S}_c . The position of O_k at timestamp j is represented as $p_j^{O_k}$ where $\mathbb{P}_{O_k} = \{p_j^{O_k} : 1 \leq j \leq t\}$.
- \mathbb{S}_{O_k} is a finite, non-empty set representing the speed of O_k at time instants of \mathcal{S}_c . The speed of O_k at timestamp j is represented as $s_j^{O_k}$ where $\mathbb{S}_{O_k} = \{s_j^{O_k} : 1 \leq j \leq t\}$.

Definition 4. A single lane l in \mathbb{L} is a tuple $\langle \mathbb{S}_l, \mathbb{P}_l \rangle$ where:

- \mathbb{S}_l is a finite, non-empty set representing the speed limit imposed by l . The speed limit of l , which the ego car is traversing at timestamp j , is represented as s_j^l where $\mathbb{S}_l = \{s_j^l : 1 \leq j \leq t\}$.
- \mathbb{P}_l is a finite, non-empty set representing the position of the closest lane boundary to the ego car. The position of l 's boundary, which the ego car is traversing at timestamp j , is represented as p_j^l where $\mathbb{P}_l = \{p_j^l : 1 \leq j \leq t\}$.

Definition 5. We define a violation $v \in \mathbb{V} = \{collision, speed, unsafeChange, fastAccl, hardBrake\}$. We elaborate on the oracles corresponding to each of these violations in Section 5.4.5.

5.4 scenoRITA

Figure 5.1 shows the overall workflow of SCENORITA. Our main goal is to create valid and effective driving scenarios that expose AV software to unique safety and comfort violations. SCENORITA achieves this goal as follows: (1) it takes as an input a set of *domain-specific constraints*, which dictates what constitutes a *valid* driving scenario (e.g., obstacles should be moving in the direction of traffic in the lane and have valid obstacle identifiers); (2) The *Scenario Generator* uses a genetic algorithm to produce driving scenarios with randomly generated but valid obstacles, following the *domain-specific constraints*. The genetic algorithm evolves the driving scenarios with the aim of finding scenarios with safety and comfort violations; (3) *Generated Scenarios Player* converts the genetic representation of scenarios (*Generated Scenarios*), from the previous step, into driving simulations where the planning output of the AV under test is produced and recorded by *Planning Output Recorder*; (4) The planning output is then evaluated by *Grading Metrics Checker* for *safety and comfort violations*; (5) When the evolution process terminates, the *Duplicate Violations Detector* inspects the violations produced by *Grading Metrics Checker* to eliminate any duplicate violations, and produces a set of *unique safety and comfort violations*. In the remainder of this section, we discuss each of these elements of SCENORITA in more detail.

5.4.1 Domain-Specific Constraints

Table 5.2 specifies the list of constraints that *Scenario Generator* should follow to ensure that the generated driving scenarios are *valid*. In this work, we define *valid scenarios* as those which contain obstacles that are (1) moving in the direction of traffic in the lane; (2) having start and end points contained within the boundaries of a fixed-size map; and (3) having

Table 5.2: A list of *Domain-Specific Constraints* that *Scenario Generator* adheres to when creating driving scenarios.

Scenario Attr.	Sub-Attr.	Description	Constraints
Ego Car	Initial Position	The start position of the ego car in the map	Initial Position should be within the map boundaries
	Final Position	The destination position of the ego car in the map	Final Position should be within the map boundaries, and there should be a valid path between the ego car's Initial and Final Position
Obstacles	ID	A unique identification number associated with each obstacle in a Scenario	Obstacles in a single Scenario have unique IDs
	Initial Position	The start position of an obstacle in the map	Initial Position should be within the map boundaries
	Final Position	The destination position of an obstacle in the map	Final Position should be within the map boundaries, and there should be a valid path between the obstacle's Initial and Final Position
	Heading	The compass direction in which the obstacle is pointing at a given time. It is expressed as the angular distance relative to north.	Heading of an obstacle has a minimum of -180° (clockwise direction), and a maximum of 180° (counter-clockwise direction), i.e., the allowed range of an obstacle Heading is $[-3.14rad - 3.14rad]$.
	Type	The type of an obstacle	An Obstacle can be one of the following values: (VEHICLE, BICYCLE, PEDESTRIAN)
	Speed	Speed of an obstacle, measured in km/hr. The obstacle type dictates the valid minimum and maximum speed of an obstacle:	
		VEHICLE	The speed of a vehicle can range from 8km/hr (e.g., parking lots) to 110km/hr (e.g., highways)
		BICYCLE	The speed of a bicycle can range from 6km/hr to 30 km/hr
		PEDESTRIAN	The speed of a pedestrian can range from 4.5 km/hr (average walking speed) to 10.5 km/hr (average running speed)
	Width	Width of an obstacle, measured in meters. The obstacle type dictates the valid minimum and maximum width of an obstacle:	
		VEHICLE	[1.5 - 2.5] in meters
		BICYCLE	[0.5 - 1] in meters
		PEDESTRIAN	[0.24 - 0.67] in meters
	Length	Length of an obstacle, measured in meters. The obstacle type dictates the valid minimum and maximum length of an obstacle:	
		VEHICLE	[4 - 14.5] in meters
		BICYCLE	[1 - 2.5] in meters
		PEDESTRIAN	[0.2 - 0.45] in meters
	Height	Height of an obstacle, measured in meters. The obstacle type dictates the valid minimum and maximum height of an obstacle:	
		VEHICLE	[1.5 - 4.7] in meters
		BICYCLE	[1 - 2.5] in meters
		PEDESTRIAN	[0.97 - 1.87] in meters
	Motion	The motion of an obstacle	An Obstacle can either be: <i>static</i> or <i>mobile</i>

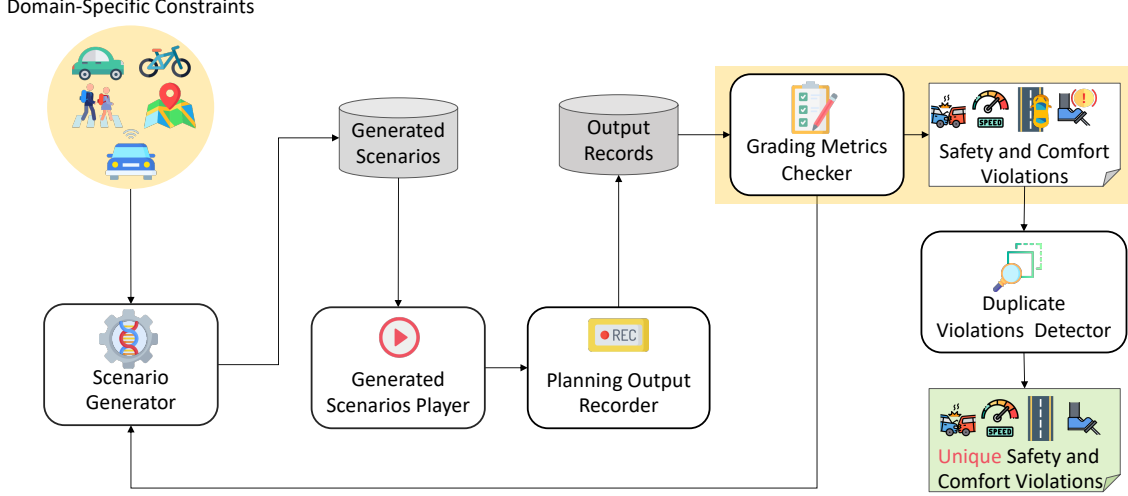


Figure 5.1: An Overview of SCENORITA

dimensions (i.e. width, height, and length) and speed that account for the obstacle type (vehicle, pedestrian, or bike). For example, the speed of a pedestrian should not exceed the average walking/running speed of a human.

When generating the initial and final position of the ego car and obstacles, *Scenario Generator* must ensure that these points are (i) within the boundaries of a fixed-size map, and (ii) have a valid path allowing the ego car and obstacles to move in the direction of traffic. To ensure that an obstacle is driving in the right direction, the heading attribute should be within a valid range. An obstacle heading is the compass direction in which an obstacle is pointing at a given time, and it is expressed as the angular distance relative to the north. An obstacle’s heading has a positive value in the counter-clockwise direction, with a maximum of 180° (π or $3.14rad$), and a negative value in the clockwise direction, with a minimum of -180° ($-\pi$ or $-3.14rad$).

Unlike prior work, we consider a wide range of obstacle-related attributes including type, size, speed, and mobility. An obstacle can be a **VEHICLE**, **BICYCLE**, or a **PEDESTRIAN**, and the type of an obstacle dictates the minimum and maximum allowed values of its size and speed. An obstacle is represented as a polygon, hence, its size is expressed in terms of the width, height, and length (i.e., Z_{O_k}) of the polygon. The ego car E is similarly represented as a polygon

based on Z_E .

To determine the maximum and minimum dimensions of a pedestrian, we followed the most recent report published by the National Center for Health Statistics (NCHS) [71], which provides the most recent anthropometric reference data for children and adults in the United States. The height of a pedestrian ranges from 0.97m (average height of a child) to 1.87m (average height of an adult aged 20+). The width (shoulder width) ranges from 0.24m to 0.67m, while the length ranges from 0.2m to 0.45m. The speed of a pedestrian can range from 4.5km/hr (average walking speed) to 10.5 km/hr (average running speed) [129].

To determine the maximum and minimum dimensions and speed for both bicycles and vehicles, we followed the size and speed regulations imposed by the Federal Highway Administration and the US Department of Transportation [91, 90]. The speed of a bicycle can range from 6km/hr to 30 km/hr, while the speed of a vehicle can range from 8km/hr (e.g., parking lots) to 110km/hr (e.g., highways).

5.4.2 Scenario Generator

Our overarching goal is to create valid and effective driving scenarios that expose AV software to safety and comfort violations. The *Scenario Generator* takes as input a set of *domain-specific constraints* (Section 5.4.1) and uses a genetic algorithm to maximize a defined set of fitness functions (representing safety and comfort violations) to guide the search for problematic scenarios. The genetic algorithm is initialized with a starting population of tests (i.e., driving scenarios). To evaluate the fitness of tests, scenario representations are transformed into driving simulations, in which navigation plans are generated based on the origin and destination of the ego car. Additionally, driving trajectories/plans are computed for the ego car based on the scenario set-up (e.g., number of obstacles, state of the obstacles, ego car start and target position, etc.). During the simulation, the driving decisions of the ego car (e.g., driving maneuvers, stop/yield decisions, acceleration) are recorded by *Planning*

Output Recorder at regular intervals to identify safety and comfort violations. A set of values such as the distance between the ego car and other obstacles, the distance between the ego car and lane boundaries, the speed of the ego car, the acceleration and deceleration of the ego car are used to compute the fitness of individuals (Section 5.4.2). These values guide the genetic algorithm when evolving test cases (i.e., driving scenarios) by recombining and mutating their attributes (Section 5.4.2). The algorithm continues to execute and evolve test cases until a user-defined ending condition is met, at that point SCENORITA returns the final test suite and stops.

Representation.

Figure 5.2(a) illustrates the genetic representation of an individual generated by SCENORITA. A set of individuals together represent a driving scenario which, in turn, represents a single test. A test suite in SCENORITA is a set of driving scenarios. An individual is represented as a vector, where each index in the vector represents a gene. The number of input genes is fixed, where the 10 genes corresponds to the following 10 attributes of a *single* obstacle O_k : ID_{O_k} ; the initial position $p_1^{O_k}$ of O_k ; the final position $p_t^{O_k}$ of O_k at the final timestamp t ; initial heading $h_1^{O_k}$; length $Z_{O_k}.len$; width $Z_{O_k}.wid$; and height $Z_{O_k}.hgt$; initial speed $s_i^{O_k}$ at timestamp j ; type T_{O_k} ; and mobility M_{O_k} . Each gene value can change (e.g., when initialized or mutated), but it still has to adhere to the valid ranges defined in Section 5.4.1.

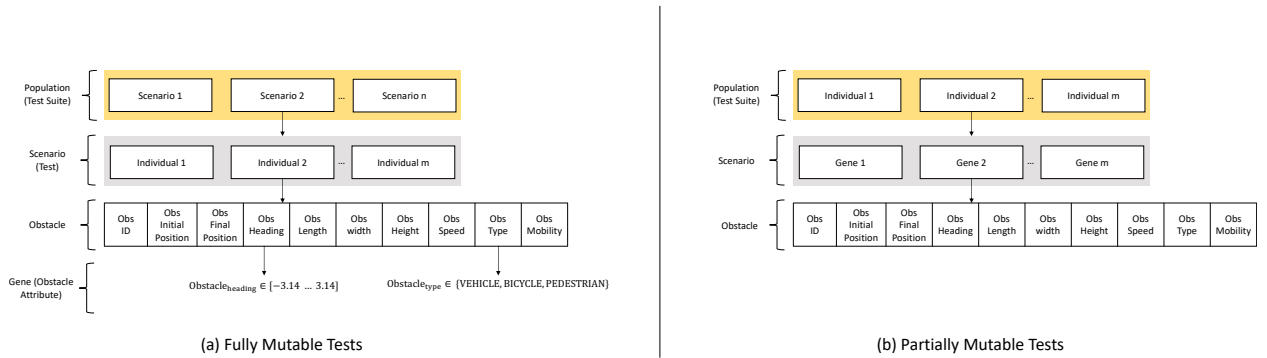


Figure 5.2: Genetic representation of: (a) fully-mutable tests in SCENORITA, and (b) partially mutable tests generated by state-of-the-art approaches.

The main contribution of our approach is its ability to represent obstacles as "individuals" instead of genes which allow them to be fully-mutable. We use a non-traditional way to relate the test suite (population) to obstacles (individuals) by introducing an intermediate component (test) that represents driving scenarios. Figure 5.2(b) shows the genetic representation of tests other meta-heuristic approaches compared to SCENORITA.

Representing *obstacles* as *individuals* allows SCENORITA to alter obstacles' attributes and states when applying search operators, hence, allowing obstacles to be *fully mutable* throughout the test generation process. Figure 5.3(b) demonstrates SCENORITA's application of a crossover operator on two individuals (i.e., obstacles) compared to how related work recombine their individuals (Figure 5.3(c)). Previous approaches [97, 67, 130], represent *obstacles* as *genes*, resulting in obstacles being *partially mutable* during recombination and mutation operators. For example, the crossover operator in *AV-Fuzzer* [130] does not alter properties of obstacles, instead it simply swaps two randomly selected obstacles in two scenarios, *Scenario B* and *Scenario D*, with a certain probability. For the remainder of the chapter, we will use SCENORITA⁺⁺ and SCENORITA⁻ to refer to *fully mutable* and *partially mutable* approaches, respectively.

By representing obstacles using individual attributes (e.g., location), as opposed to just as a gene in the case of SCENORITA⁻, SCENORITA⁺⁺ must address the challenge of ensuring the creation of valid obstacle trajectories. To that end, SCENORITA⁺⁺ includes logic that allows it to check whether there is a valid path between the newly-generated start and end locations of an obstacle. If the recombination operator introduces an invalid path, then SCENORITA⁺⁺ generate new locations for an obstacle until a valid one is found. This process allows SCENORITA⁺⁺ to represent an obstacle's individual attributes, such as the start and end locations, as genes while preventing the invalid genes that may be produced due to the application of search operators. Our evaluation results in Section 5.5 shows that SCENORITA⁺⁺ results in 23.47% more effective and unique violations compared to

SCENORITA⁺⁺.

Fitness Evaluation.

In each generation, individuals are assessed for their fitness, with respect to the search objective, to be selected to pass on their genes. SCENORITA determines the fitness of an individual by evaluating how close they are in terms of causing safety or comfort violations. This is measured by calculating an individual i 's fitness using a function $f_v(i)$ with respect to a safety and comfort violation v . Recall from Section 5.1 that SCENORITA considers 5 safety and comfort violations based on the grading metrics defined by Apollo's developers [19] and, thus, represents violation constructs and thresholds used by professional AV developers. Three of these metrics assess driving scenarios for traffic and road safety (collision detection, speeding detection, and unsafe lane change), while the remaining two metrics assess a rider's comfort (fast acceleration and hard braking).

The fitness of an individual i is determined as follows:

$$F(i) = \left(f_{collision}(i), f_{speed}(i), f_{unsafeChange}(i), f_{fastAccl}(i), f_{hardBrake}(i) \right) \quad (5.1)$$

Recall that $v \in \mathbb{V} = \{collision, speed, unsafeChange, fastAccl, hardBrake\}$ (Definition 5), hence, $F(i)$ aims to maximize the number of violations. In the remainder of this section, we define $f_v(i)$ in more detail.

Collision Detection. In the context of collision detection, effective tests are those which cause the ego car to collide with other obstacles. Therefore, SCENORITA uses as a fitness function $f_{collision}$ (Equation 5.2), which rewards tests that cause the ego car to move as close as possible to other obstacles. Given a simulated scenario with a maximum duration of t , a set of positions \mathbb{P}_E for the ego car E , and a set of positions for the k^{th} obstacle in a scenario

defined as $\mathbb{P}_{O_k} = (p_1^{O_k}, p_2^{O_k}, \dots, p_t^{O_k})$, we define $f_{collision}$ as:

$$f_{collision}(i) = \min\{D_c(p_j^E, p_j^{O_k}) \mid 1 \leq j \leq t \wedge p_j^E \in \mathbb{P}_E \wedge p_j^{O_k} \in \mathbb{P}_{O_k}\} \quad (5.2)$$

$D_c(p^E, p^{O_k})$ is the shortest distance between the position of a given obstacle and the ego car at a given time. $f_{collision}$ captures the intuition that tests causing the ego car to drive closer to other obstacles (i.e., have a minimum distance between the ego car and an obstacle) are more likely to lead to a collision.

Speeding Detection. We use a fitness function f_{speed} (Equation 5.3), which rewards tests that cause the ego car to exceed the speed limit of the current lane. Given a simulated scenario with a maximum duration t , the speed profile \mathbb{S}_E of the ego car E , and a set of speed limits imposed by lanes of which the ego car traversed \mathbb{S}_L , we define f_{speed} as:

$$f_{speed}(i) = \min\{D_s(s_j^l, s_j^E) \mid 1 \leq j \leq t \wedge s_j^E \in \mathbb{S}_E \wedge s_j^l \in \mathbb{S}_L\} \quad (5.3)$$

s_j^E and s_j^l represent the ego car speed and the speed limit of the lane in which the ego car is travelling at timestamp j , respectively. Furthermore, $D_s(s^l, s^E)$ is the difference between the speed limit imposed by a given lane and the current speed of the ego car. f_{speed} captures the intuition that as the ego car approaches the speed limit of a given lane it is more likely to result in speed violations.

Unsafe Lane Change. A lane change is defined as a driving maneuver that moves a vehicle from one lane to another, where both lanes have the same direction of travel. We primarily focus on the *duration* the ego car spends travelling at the boundary of two lanes while changing lanes. We define a *safe* lane-change duration as δ_{safe} . We define a fitness function $f_{unsafeChange}$ (Equation 5.4), which rewards tests that cause the ego car to spend more than

δ_{safe} driving at the boundary of two lanes. Given a simulated scenario with lane-change durations \mathbb{C}_E for ego car E , and a safe lane-change duration δ_{safe} , we define $f_{unsafeChange}$ as:

$$f_{unsafeChange}(i) = \begin{cases} \max(c_j^E) & D_u(p_j^E, p_j^l) = 0 \mid 1 \leq j \leq t \\ \min(D_u(p_j^E, p_j^l)) & otherwise \end{cases} \quad (5.4)$$

$D_u(p_j^E, p_j^l)$ is the shortest distance between the position of the ego car and a lane boundary at time j . $D_u(p_j^E, p_j^l) = 0$ indicates that the ego car is driving at the boundary of two lanes, while $c_j^E \in \mathbb{C}_E$ represents the duration an ego car spends driving between two lanes. $f_{unsafeChange}$ captures the intuition that tests causing the ego car to spend longer periods of time driving on lane boundaries are more likely to result in an unsafe lane change violation. If the ego car is not driving on lane boundaries (i.e., when $D_u(p_j^E, p_j^l) \neq 0$), then $f_{unsafeChange}$ rewards tests that cause the ego car to drive as close as possible to lane boundaries (i.e., minimizes the distance between the ego car and lane boundaries).

Fast Acceleration. We use a fitness function $f_{fastAccl}$ (Equation 5.5), which rewards tests that cause the ego car to accelerate too fast, potentially inducing motion sickness. Given a simulated scenario with a maximum duration t and the acceleration profile \mathbb{A}_E for the ego car E , we define $f_{fastAccl}$ as:

$$f_{fastAccl}(i) = \max\{a_j^E \mid 1 \leq j \leq t \wedge a_j^E \in \mathbb{A}_E\} \quad (5.5)$$

$a_j^E \in \mathbb{A}_E$ represents the acceleration of the ego car E at timestamp j . $f_{fastaccl}$ aims to maximize the acceleration of E to induce a motion sickness violation.

Hard Braking. We use a fitness function $f_{hardBrake}$ (Equation 5.6), which rewards tests that cause the ego car to brake too hard (i.e., brake suddenly in a manner that induces motion sickness). Given a simulated scenario with a maximum duration t , the acceleration profile for

the ego car defined as \mathbb{A}_E , we define $f_{hardBrake}$ as:

$$f_{hardBrake}(i) = \min\{a_j^E \mid 1 \leq j \leq t \wedge a_j^E \in \mathbb{A}_E\} \quad (5.6)$$

$a_j^E \in \mathbb{A}_E$ represents the deceleration of the ego car. $f_{hardBrake}$ captures the intuition that tests which cause the ego car to decelerate too fast can result in hard braking.

Search Operators.

SCENORITA evolves driving scenarios by applying search operators, which mutate and recombine the scenario attributes according to certain probabilities. In this section, we provide a detailed explanation of these search operators.

Selection. SCENORITA uses the **Non-dominated Sorting Genetic Algorithm** selection (NSGA-II [77]) for breeding the next generation. NSGA-II is an effective algorithm used for solving multi-objective optimization problems (i.e., problems with multiple conflicting fitness functions) and further aims to maintain diversity of individuals.

NSGA-II starts by sorting a set of individuals based on a *non-dominated* order. In a multi-objective problem, an individual i_1 is said to *dominate* another individual i_2 if (1) i_1 is no worse than i_2 for **all** objective functions (e.g., collision detection, speeding detection, etc.), and (2) i_1 is strictly better than i_2 in at least one objective. Once the non-dominated sort is complete, a *crowding distance* is assigned to every individual in a given scenario. A *crowding distance* measures how close individuals are to each other; a large average crowding distance will result in better diversity in the population. Once the crowding distance is assigned, parent individuals are selected to produce offspring based on the fitness and crowding distance; an individual is selected if its order rank is less than the other, or if the crowding distance is greater than the other. Only the best N individuals are selected, where N is the population size.

The intuition behind using NSGA-II selection is threefold: (1) it uses an elitist principle, i.e., the most elite individuals in a scenario are given the opportunity to be reproduced so their genes can be passed on to the next generation; (2) it uses an explicit diversity-preserving mechanism (i.e., crowding distance), which maintains the diversity of driving scenarios in SCENORITA; and (3) it emphasizes the non-dominated solutions.

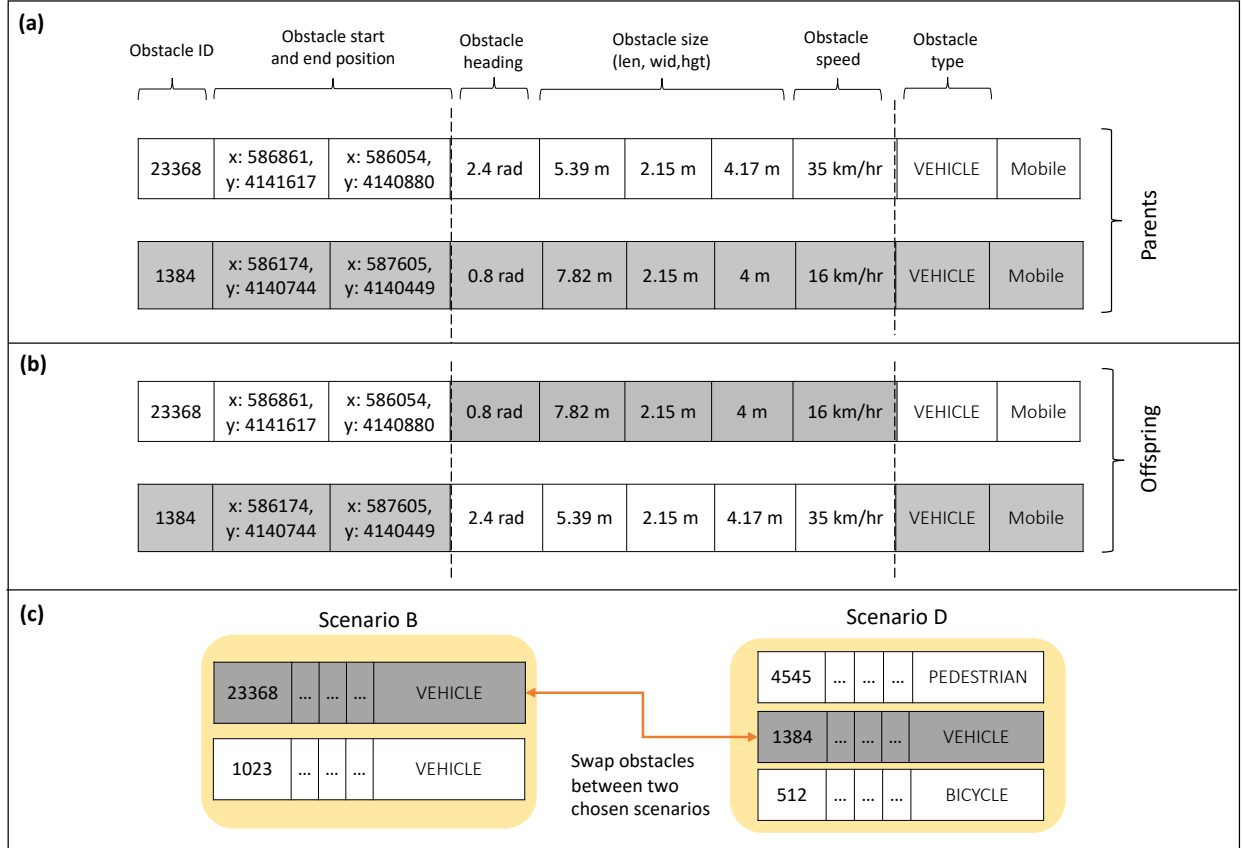


Figure 5.3: (a) two individuals *before* a crossover, (b) the same individuals *after* a crossover for SCENORITA, and (c) how crossover is applied in prior work [130]

Crossover. This operator selects two individuals from a given scenario and creates superior offspring by mixing their parents' genetic makeup. SCENORITA uses a two-point crossover strategy, where two crossover points are picked randomly from the mating individuals (i.e., parents) and the genes between the two points are swapped. Figure 5.3 illustrates the application of the two-point crossover operator on two sample individuals; the two individuals are modified in place and both keep their original length. We opt for the two-point crossover strategy since it maintains the length of individuals, in addition to increasing the extent of

disruption in their original values.

Obstacle ID	Obstacle start and end position		Obstacle heading	Obstacle size (len, wid, hgt)			Obstacle speed	Obstacle type	
23368	x: 586861, y: 4141617	x: 586054, y: 4140880	2.4 rad	4 m	1.5 m	1.5 m	50 km/hr	VEHICLE	Mobile
1384	x: 586174, y: 414074	x: 587605, y: 414044	0.8 rad	0.45 m	0.67 m	1.87 m	10 km/hr	PEDESTRIAN	Mobile
23368	x: 586861, y: 4141617	x: 587605, y: 414044	0.8 rad	0.45 m	0.67 m	1.87 m	10 km/hr	VEHICLE	Mobile
1384	x: 586174, y: 414074	x: 586054, y: 4140880	2.4 rad	4 m	1.5 m	1.5 m	50 km/hr	PEDESTRIAN	Mobile

Figure 5.4: An example of a crossover that produced individuals with invalid attributes (highlighted in red)

Crossover may produce invalid scenario configurations. For example, after crossover of a vehicle’s attributes with those of a pedestrian (Figure 5.4), some obstacle attributes produced in the offspring may violate the speed and size constraints in Section 5.4.1, such as having a pedestrian’s speed changed from **10** km/hr (a valid running speed for a pedestrian) to **50** km/hr (an unrealistic speed for a pedestrian). When such a case is detected, SCENORITA replaces the violated obstacle attributes with randomly generated values that fall within the valid ranges described in Section 5.4.1.

Mutation. SCENORITA applies the mutation operator to driving-scenarios in two forms: (1) it mutates individuals in a single scenario; and (2) it applies mutation operators across scenarios. The first type of mutation, randomly replaces genes in individuals with new ones, where the newly generated values follow the constraints defined in Section 5.4.1. For example, the mutation operator can change the speed of a vehicle from **35** km/hr to **50** km/hr. This type of mutation does not change the number of individuals in a single scenario. The second

type of mutation aims to diversify the *number* of individuals in a given scenario \mathcal{S}_c^A by (1) adding the fittest obstacle from another randomly selected scenario \mathcal{S}_c^B to \mathcal{S}_c^A , or (2) removing the least fit individual in \mathcal{S}_c^A .

Figure 5.5(b) shows the addition of a new individual (highlighted in green) to *Scenario A*; while Figure 5.5(c) shows the removal of the least fit individual (highlighted in yellow) from *Scenario A*.

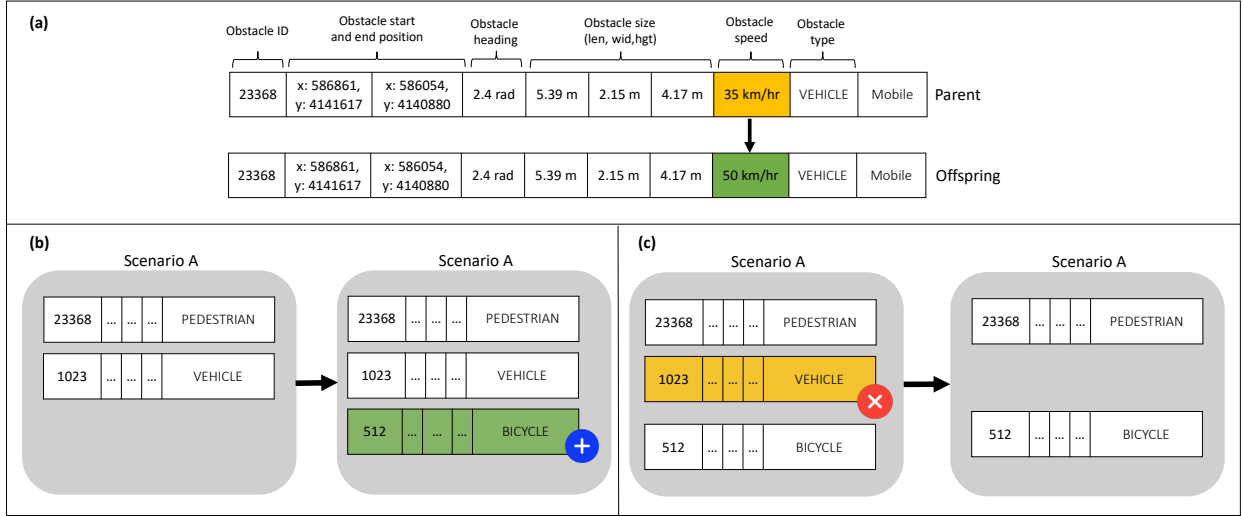


Figure 5.5: (a) mutating a gene in a *single* individual, (b) mutating a scenario by adding a fit individual from another scenario, and (c) mutating a scenario by removing the worst individual.

5.4.3 Generated Scenarios Player

Converting the genotypic representation of tests to driving simulations is the key task of *Generated Scenarios Player*. Our approach uses code templates to generate the necessary simulation code, which instantiates the obstacles in the driving simulation, places the ego car in the starting position, and sets the target position of the ego car. *Generated Scenarios Player* ensures that the ego car does not start in the middle of an intersection where it is too difficult for the ego car to accelerate fast enough to avoid oncoming traffic. Additionally, starting from and having to accelerate from zero in an intersection causes unrealistic unsafe lane changes because an intersection has multiple lanes. Once the simulation code is ready,

Generated Scenarios Player starts the driving simulator, where the planning module computes the driving trajectory taking into consideration various aspects of the vehicle and elements of its environment (e.g., distance to lane center, smoothness of the trajectory).

5.4.4 Planning Output Recorder

The behavior of the ego car in the driving simulation (Section 5.4.3) is recorded by the *Planning Output Recorder*, which stores the ego car’s driving behaviours in a record file. The recorded output enables *Grading Metrics Checker* to identify and report the occurrences of *safety and comfort violations*. These violations are checked when the driving scenario ends; hence, it does not halt the driving simulation after observing the first violation; instead, tests continue until the end of the scenario. This approach balances the cost of running expensive simulations with the benefit of collecting as many violations as possible. SCENORITA uses the output records for reporting violations, and evaluating the fitness of tests which guides the evolution process in Section 5.4.2. Additionally, stored records enable us to replay scenarios with reported violations after SCENORITA ends; this allows us to verify the correctness of generated tests (Section 5.5.2), and to closely analyze them along with their underlying causes. We make these record files available for researchers and practitioners to reuse, replicate, or analyze them in future work [1].

5.4.5 Grading Metrics Checker

Previous work [130, 97, 96, 60, 47, 67, 184] considers a limited number of test oracles, mainly consisting of one test oracle per work (either collision detection or lane keeping). The limited use of test oracles found in such techniques ignores important safety and comfort issues (e.g., driving between lanes for too long or causing motion sickness) and provides significantly less insight into the testing of industry-grade AVs. Unlike previous work, we consider 5 test oracles based on grading metrics defined by Apollo’s developers [19]. These grading metrics test different aspects of AVs, ranging from traffic and road safety to a rider’s comfort. In the

remainder of this section, we describe each grading metric in detail along with the definition of its corresponding test oracle.

The **Collision Detection** oracle checks if the ego car reaches its final destination without colliding with other obstacles. The test oracle’s passing condition (i.e., not resulting in a violation) for collision detection is defined as follows:

$$\forall_{min_avoid \leq j \leq t} (D_c(p_j^E, p_j^{O_k}) \neq 0 \vee (D_c(p_j^E, p_j^{O_k}) \leq 0 \mid s_j^E \leq th_{min_speed} \wedge collision^{type} = \text{“rear-end”})) \quad (5.7)$$

where t is the total duration of the scenario, and $D_c(p_j^E, p_j^{O_k})$ is a function that calculates the *shortest distance* between the position of the ego car p_j^E and the position of the k^{th} obstacle $p_j^{O_k}$ at timestamp j . The distance is measured, in meters, between the closest two points between the ego car’s polygon and an obstacle polygon. If function D_c returns a non-zero distance in meters between the ego car and any other obstacle, this indicates a passing condition, i.e., a collision avoidance. If a collision does occur (i.e., D_c returns a distance equal to or less than zero), we exclude rear-end collisions when $s_j^E \leq th_{min_speed}$. This design effectively eliminates cases where the ego car cannot avoid being hit from behind when driving at a slow speed (e.g., when stopping or stopped at a red light). Otherwise, an excessive amount of avoidable collisions are detected as violations, which the ego car cannot be reasonably responsible for. Lastly, we do not detect collisions until a specified time, i.e., min_avoid , which is the minimum amount of time the ego car needs to avoid a collision. min_avoid is set to 3 seconds as we determined experimentally.

The **Speeding Detection** oracle checks if the ego car reaches its final destination without exceeding the speed limit. The test oracle’s passing condition (i.e., not resulting in a violation)

for speeding detection is defined as follows:

$$\bigvee_{1 \leq j \leq t} D_s(s_j^l, s_j^E) \leq \sigma_{safe} \quad (5.8)$$

where t is the total duration of the scenario, and $D_s(s_j^l, s_j^E)$ is a function that calculates the difference between the ego car's speed s_j^E and the speed limit of the current lane s_j^l at timestamp j . σ_{safe} represents the allowed threshold for an ego car to drive *above* the current speed limit. We allow the ego car to exceed the current speed limit by a maximum of **8** km/hr, anything above that is considered a speed violation. We allow some degree of driving above the speed limit, since it can be unsafe for the ego car to drive below or at the speed limit in certain conditions [22] (e.g., driving at the speed limit when other cars are going much faster can be dangerous).

The **Unsafe Lane-Change** oracle checks if the ego car reaches its final destination without exceeding a time limit δ_{safe} when changing lanes. Recall from Section 5.4.2, that δ_{safe} represents a safe lane-change duration, which averages at 5 seconds [92]. The test oracle's passing condition (i.e., not resulting in a violation) for unsafe lane change is defined as follows:

$$\bigvee_{1 \leq j \leq t} c_j^E \leq \delta_{safe} \quad (5.9)$$

where t is the total duration of the scenario, and c_j^E represents the duration an ego car spends driving between two lanes at timestamp j . If c_j^E at a given time exceeds δ_{safe} , this indicates the occurrence of an unsafe lane change.

The **Fast Acceleration** oracle checks if the ego car reaches its final destination without causing a rider's discomfort by accelerating too fast. The test oracle's passing condition for

fast acceleration is defined as follows:

$$\bigvee_{1 \leq j \leq t} a_j^E \leq \alpha_{fast} \quad (5.10)$$

where t is the total duration of the scenario, and a_j^E is the acceleration of the ego car at timestamp j . α_{fast} represents the maximum acceleration allowed for an ego car before it violates a rider’s comfort. We allow the ego car to accelerate to a maximum of 4 m/s^2 , a threshold utilized in prior work [59] and set by Apollo developers [19].

The **Hard Braking** oracle checks if the ego car reaches its final destination without causing a rider’s discomfort by braking suddenly and excessively. The test oracle’s passing condition for hard braking is defined as follows:

$$\bigvee_{1 \leq j \leq t} a_j^E \geq \alpha_{hard} \quad (5.11)$$

where t is the total duration of the scenario, and a_j^E is the acceleration of the ego car at timestamp j . α_{hard} represents the minimum acceleration allowed for an ego car before it violates a rider’s comfort. We allow the ego car to decelerate to a minimum of -4 m/s^2 , a threshold used in prior work [59] and set by Apollo’s developers [19].

5.4.6 Duplicate Violations Detector

One of the challenges of scenario-based testing is the possibility of producing driving scenarios with *similar* violations. To improve the effectiveness of test generation, the *Duplicate Violations Detector* automates the process of identifying and eliminating duplicate violations; it achieves this by using an unsupervised *clustering* technique [86] to group driving scenarios, with similar violations, according to specific *features*.

The set of features used by the clustering algorithm are extracted from the recorded files (Section 5.4.4). For a **collision** violation, the *Duplicate Violations Detector* extracts 8 features

at time t_c , where t_c indicates the first timestamp at which a collision occurs. These features include the location of the ego car $p_{t_c}^E$; ego car’s speed $s_{t_c}^E$; ego car’s heading $h_{t_c}^E$; $collision^{type}$, which indicates where a collision occurs in respect to the ego car (e.g., “rear-end”, “front”, “left”, etc.); the type of the obstacle (O_k) that collided with the ego car T_{O_k} ; the obstacle’s size Z_{O_k} ; obstacle’s speed at collision time $s_{t_c}^{O_k}$; and obstacle’s heading $h_{t_c}^{O_k}$.

For the remaining violations, we extract their respective features at times t_s , t_u , t_f , and t_h , which correspond to the first timestamp a **speeding**, **unsafe lane change**, **fast acceleration**, and **hard braking** occurs, respectively. These features include the ego car E ’s location at a violation time p^E , the speed s^E of E , the heading h^E of E , the length of time for which a violation lasts (*duration*), and the violation *value*.

Table 5.3: The set of features, selected for each violation type, and used by the *Duplicate Violation Detector* to cluster similar violations together.

Grading Metric	Extracted Features
Collision Detection	$\{p_{t_c}^E, s_{t_c}^E, h_{t_c}^E, collision^{type}, T_{O_k}, Z_{O_k}, s_{t_c}^{O_k}, h_{t_c}^{O_k}\}$
Speeding Detection	$\{p_{t_s}^E, s_{t_s}^E, h_{t_s}^E, duration, value_{t_s}\}$
Unsafe LaneChange	$\{p_{t_u}^E, s_{t_u}^E, h_{t_u}^E, duration_{t_u}\}$
Fast Acceleration	$\{p_{t_f}^E, s_{t_f}^E, h_{t_f}^E, duration, value_{t_f}\}$
Hard Braking	$\{p_{t_h}^E, s_{t_h}^E, h_{t_h}^E, duration, value_{t_h}\}$

Given the extracted representation of driving violations in Table 5.3, *Duplicate Violations Detector* clusters driving scenarios with similar violations into groups. For the clustering itself, we chose DBSCAN (i.e., density-based spatial clustering of applications with noise) [86], since it is more suited for spatial data. We also experimented with k-means, which resulted in clusters of undesired structure and quality. We avoided the use of hierarchical clustering [115] due to its computationally expensive nature.

Existing work has suggested using clustering techniques to automatically categorize traffic scenarios or driving behaviours [105, 136, 176, 125, 182]. These approaches are geared towards clustering real-time, multi-trajectory, and multivariate time series data into similar

driving encounters or scenario types. Unlike these techniques, SCENORITA aims to eliminate duplicate violations by clustering scenarios with violations. Hence, *Duplicate Violations Detector* only requires a carefully-selected, smaller number of features involving just a few time frames in a scenario.

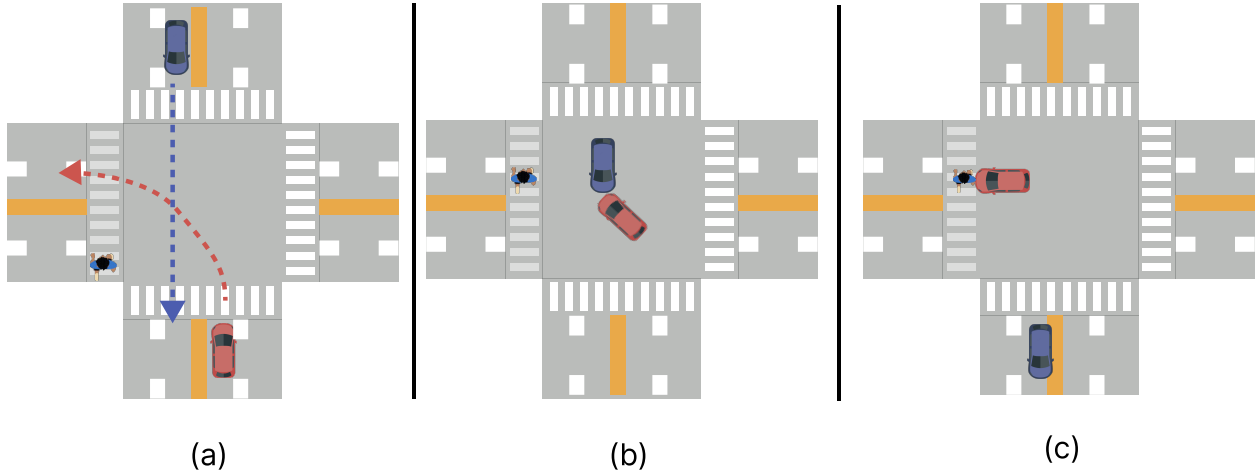


Figure 5.6: An illustration of one of five supported driving scenarios in AutoFuzz. (a) The ego car (in red) starts at a fixed location then turns left at a signalized junction, while another vehicle (in blue) crosses the intersection from the other side and a pedestrian crosses the street. (b) The ego car turns left and collides with an incoming car (in blue). (c) The ego car turns left and collides with a pedestrian crossing the street.

Duplicate elimination in AsFault [97] is based on similar road segments, where the similarity between roads is calculated using the Jaccard index. In AV-Fuzzer [130], duplicate elimination is based only on obstacles’ trajectories, where the similarity between trajectories is calculated using the Euclidean distance.

AutoFuzz [184] detects duplicate traffic violations using a learning-based seed selection and mutation strategy. Recall from Table 5.1 that AutoFuzz generates a limited number of scenario types (maximum of 5). The paper claims many violations as unique. However, after manual inspection of those violations, it is evident that they are, in fact, extremely similar, but are detected as unique by setting a low threshold for uniqueness in their corresponding experiments [184]. For example, the ego car in one of the supported scenarios by AutoFuzz (depicted in Figure 5.6(a)) collides with either incoming traffic (i.e., either a car or truck)

as shown in Figure 5.6(b), or with a pedestrian crossing the street Figure 5.6(c). Due to (1) the trajectories for both the ego car and obstacles in an AutoFuzz scenario remain fixed throughout evolution and (2) collisions occur approximately in the same location (e.g., either with incoming traffic in the middle of the intersection or the pedestrian crossing the street), AutoFuzz counts these slight changes in violations found in a scenario as unique from each other. For example, up to hundreds of slight variations of Figure 5.6(b) are counted as different violations, similarly for Figure 5.6(c), which results in a large overcounting of unique violations. Note that AutoFuzz is, at the point of this chapter’s writing, not published in a peer-reviewed venue. However, it is the only original implementation of an AV testing technique that works on a production-grade AV system.

5.5 Evaluation

In order to empirically evaluate SCENORITA, and to understand how its configuration affects the quality of generated tests, we investigate the following research questions:

RQ1: How accurate are the driving scenarios generated by SCENORITA?

RQ2: How effective are SCENORITA’s generated driving scenarios at exposing AV software to safety and comfort violations?

RQ3: What is the runtime efficiency of SCENORITA’s generated tests and oracles?

RQ4: To what extent does SCENORITA eliminate duplicate violations?

5.5.1 Experiment Settings

Our extensive evaluation consists of executing 31,413 virtual tests on Baidu Apollo. For this reason, we conducted our experiments on three machines: 4 AMD EPYC 7551 32-Core Processor (512GB of RAM), 1 AMD EPYC 7551 32-Core Processor (256GB of RAM), and 1 AMD Opteron 64-core Processor 6376 (256GB RAM) all running Ubuntu 18.04.5. In the current implementation of SCENORITA, we focus our efforts on testing Baidu Apollo 6.0 [15],

an open-source and production-grade AV software system that supports a wide variety of driving scenarios and explicitly aims for both safety and rider’s comfort.

We use Apollo’s simulation feature, `Sim-Control`, to simulate driving scenarios. `Sim-Control`[42] does not simulate the *control* of the ego car; instead, the ego car acts on the planning results.

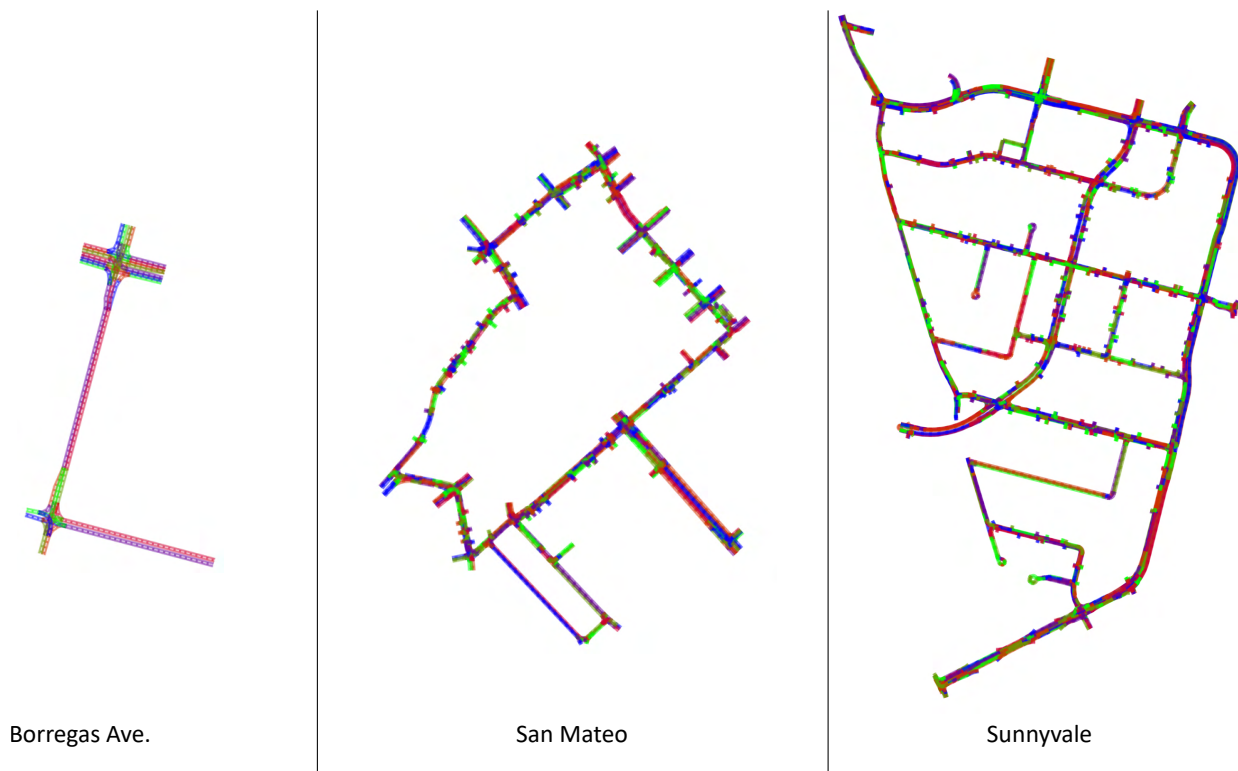


Figure 5.7: Three HD maps used by SCENORITA: **Borregas Ave** is a small map of a city block in Sunnyvale with **60** lanes and a total length of 3 km; **San Mateo** is a medium map with **1,305** lanes and a total length of 24 km; **Sunnyvale** is a large map consisting of **3061** lanes, with a total length of 107 km.

We configured SCENORITA to generate driving scenarios for 3 high-definition maps of cities/street blocks located in California: Sunnyvale is a large map consisting of 3,061 lanes, with a total length of 107 km; San Mateo is a medium map with 1,305 lanes and a total length of 24 km; and Borregas Ave, which is a small map of a city block in Sunnyvale with 60 lanes and a total length of 3 km. The three maps consist of various types of road curvature (e.g., straight, curved, intersections) and different types of lanes (e.g., highways, city roads, bike lanes, etc.). California is at the center of AV research, and one of the main deployment

grounds for AV; Waymo, GM Cruise, and 60 other companies have obtained commercial licenses for testing AVs in CA [40]. As a result, these maps are highly representative of real-world AV driving scenarios with a wide variety of diverse environmental elements.

We believe that bugs in AV software can be exposed not only by placing certain obstacles with certain attributes near the ego car, but also generating tests that cover as many different parts of the map, and subsequently different road setup (intersection, u-turn, multi-lane roads, etc.), as possible. For example, A large-scale study was conducted on 60,000 collisions in Orange County between 2010 and 2018, to determine which intersections pose the most risk for drivers in Orange County [43]. This study ranked certain intersections of cities in Orange County based on a Crash Risk Index (CRI) score—a composite score that weighs the volume of collisions and severity of injuries. The study found out that certain intersections, such as Alicia Parkway and Jeronimo Road—the only Mission Viejo crossing to make the list—saw the most injuries and the third-highest number of crashes during the study period. Similarly, in October 2021, a Pony.ai vehicle operating in autonomous mode hit a street sign on a median in Fremont, California, prompting California to suspend the company’s driverless testing permit; Pony.ai said that the crash occurred less than 2.5 seconds after the automated driving system shut down. It said “in very rare circumstances, a planning system diagnostic check could generate a false positive indication of a geolocation mismatch” [46]. These studies and incidents further emphasize the need to consider as many different parts of the map as possible, as different road setups might expose more bugs or result in more violations. Figure 5.7 showcases the scale and degree of complexity between the three maps used in our experiments. We further discussed in Section 5.5.3 the impact of each map on the found violations, and how our fully-mutable version of SCENORITA was more effective in finding violations, especially when running tests on larger and more complex maps.

We did not compare directly against implementations of previous work [130, 184, 97, 67] for two main reasons: (1) SCENORITA encodes 5 types of violations in its fitness function while

prior approaches encode a maximum of one violation type per work. To fairly compare against previous approaches, we would have to make significant changes to them to account for all five violation types. (2) Prior work was either conducted on unavailable or non-production-grade AV software. Unlike these approaches, we implemented SCENORITA to generate tests for open-source, production-grade AV software.

To evaluate the effectiveness of our full approach (SCENORITA⁺⁺), we compare it with a partially mutable representation of SCENORITA (SCENORITA⁻) and a random version of our approach that leverages SCENORITA’s *domain-specific constraints* and randomly generates obstacles (RANDOM). All three representations of SCENORITA contain the same components described in Figure 5.1, except for the *Scenario Generator*, which dictates how driving scenarios are generated. SCENORITA⁺⁺ and SCENORITA⁻ use a genetic algorithm to guide the test generation by maximizing unique violations. While SCENORITA⁺⁺ represents obstacles as individuals allowing them to be *fully mutable*, SCENORITA⁻ represents obstacles as genes, resulting in them being *partially mutable*. Both SCENORITA⁺⁺ and SCENORITA⁻ use the same search operators algorithms described in Section 5.4.2. The *Scenario Generator* in RANDOM does not contain any genetic algorithm, and it produces driving scenarios by randomly generating obstacles.

We evolved populations of 50 scenarios per generation, each with a minimum of 1 obstacle per scenario and a maximum of 70 obstacles. We configured the maximum scenario duration to be 30 seconds and stopped scenario generation after 12 hours. We used the crossover operator with a probability of 0.8 and mutated single individuals with a probability of 0.2. Mutating a scenario by either adding a new obstacle from another scenario or removing an obstacle was performed with a probability of 0.1 each. we followed the guidelines in [55, 54]—which suggests that standard parameter settings are usually recommended—leading us to use default settings in DEAP-1.3 [94], the framework used in our search-based implementation. We run each *representation* (SCENORITA⁺⁺, SCENORITA⁻ and RANDOM) on all maps (Borregas,

San Mateo and Sunnyvale), and repeated each experiment 5 times resulting in a total of 45 experiments and 540 hours of test executions.

5.5.2 RQ1: Accuracy of Generated Scenarios

We manually verify SCENORITA’s accuracy since it is not possible to verify it otherwise, due to the fact that the tests provided by Apollo’s interfaces only check whether the configuration of modules are correct. Moreover, there is no available ground truth that enables us to evaluate the accuracy of our generated scenarios. To that end, we utilize Apollo Dreamview [12], a web-based application that visualizes the ego car’s driving behaviour. Dreamview’s interface allows users to view the current behaviour of the ego car along with surrounding obstacles (i.e., pedestrians, bikes, and cars), the ego car’s speed in km/h and its acceleration/braking percentage (at the top-right corner of the screen), information about the lanes (i.e., speed limit imposed by lanes), etc.

Recall from Section 5.4.4, that the *Planning Output Recorder* is responsible for storing the ego car’s driving behaviours in record files, that can be replayed on Dreamview. Furthermore, the *Grading Metrics Checker* (Section 5.4.5)—while evaluating scenarios for safety and comfort violations—collects some metadata related to such violations. For example, if a collision is reported for a scenario, we collect information related to the obstacle that collided with the ego car (i.e., T_{O_k} and ID_{O_k}), the type of collision (i.e., rear-end, left, right), the location of the collision on the map, etc. To this end, three of the authors painstakingly and carefully evaluated the accuracy of generated scenarios by (1) replaying these scenarios on Dreamview and (2) comparing any observed missing or existing violations (e.g., whether the ego car collided with obstacle o in lane l , or if the speed of the ego car exceeded the speed limit imposed by a lane) with those reported by the *Grading Metrics checker*.

We verify the accuracy (missing or existing violations) of 4,426 out of 19,247 (i.e. 23%) of all generated scenarios. Since the manual verification takes 1.5-2 minutes per scenario, the total

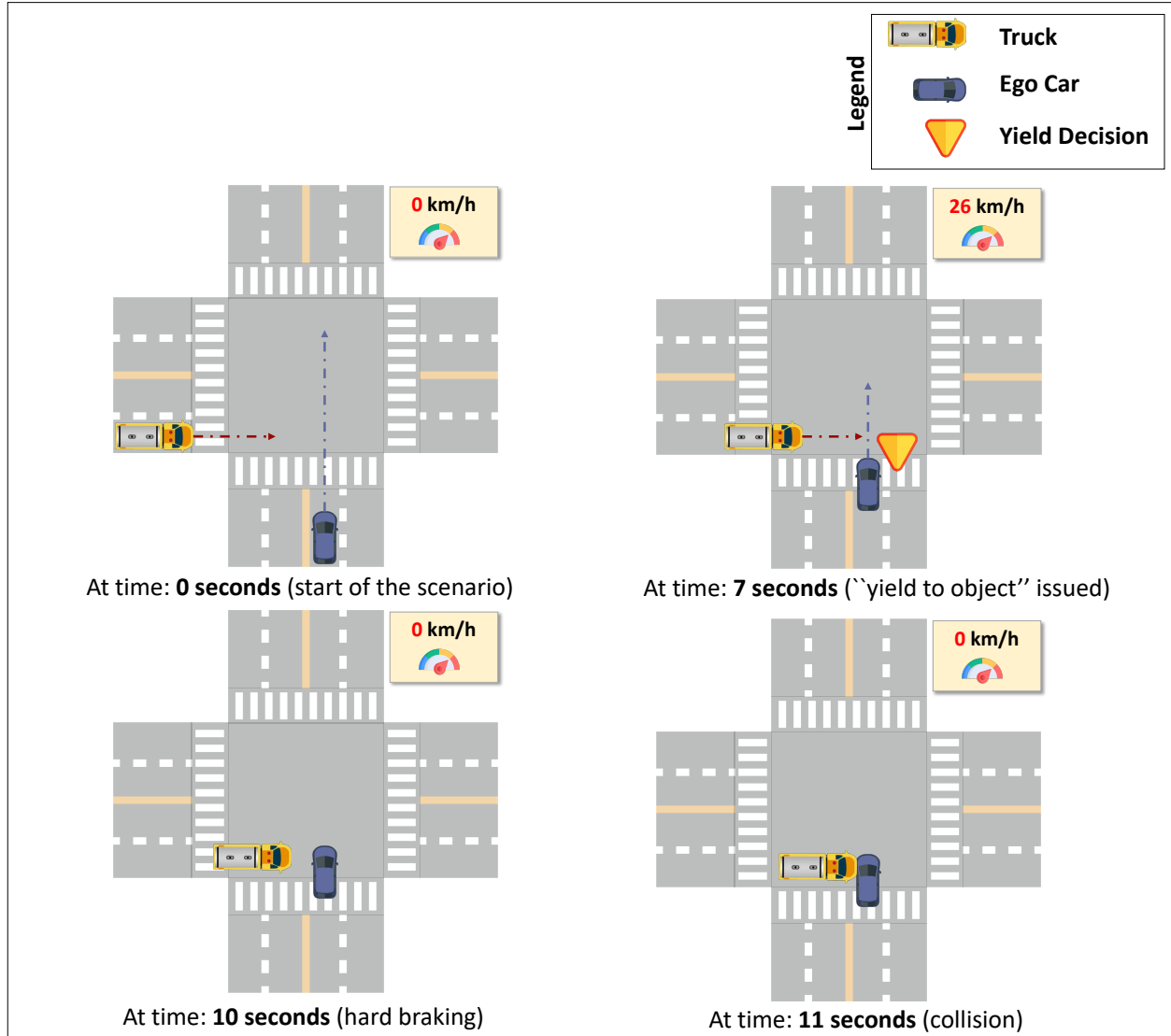


Figure 5.8: An example of one scenario generated by SCENORITA with two reported violations: collision and hard braking. This image is obtained from Dreamview, the visual simulator of Apollo.

time to verify 23% of all generated scenarios is 135 hours. Due to the time-consuming nature of manually verifying the accuracy of generated scenarios on Dreamview, we randomly sample a total of 4,426 out of 19,247 reported violations (23%), and visualize them on Dreamview to verify their correctness. To keep our sample representative of all violations, we select sample scenarios with (i) a single violation (e.g., collision only), (ii) multiple violations (up to 4 per scenario), (iii) different combinations of violations per scenario (e.g., collision and speed, collision and hard braking), and (iv) scenarios selected from different times during the

test generation process (i.e., scenarios were selected from the first few hours, after 6 hours, and the last few hours). All of the scenarios in our sample are accurate, i.e., the reported violations are consistent with the violations observed on Dreamview. Details of the verified scenarios are available in [1]. As a result, we find that:

Finding 1: Our manual verification of 23% of reported violations show that SCENORITA is able to generate scenarios with highly accurate safety and comfort violations.

Table 5.4: The number of all violations (All Viol.) reported by SCENORITA⁺⁺, SCENORITA⁻, and RANDOM, along with the total number of unique violations (Unique Viol.), and the percentage of duplicate violations eliminated (Elim. (%)). We highlight cells with the best reported results in grey.

	scenoRITA ⁺⁺			scenoRITA ⁻			Random		
	All Viol.	Uniq. Viol.	Elim. (%)	All Viol.	Uniq. Viol.	Elim. (%)	All Viol.	Uniq. Viol.	Elim. (%)
Collision	411	386	6.08%	328	246	25.00%	305	264	13.44%
Speed	25	21	16.00%	24	18	25.00%	27	18	33.33%
Unsafe Lane Change	497	291	41.45%	506	275	45.65%	509	269	47.15%
FastAccl	212	132	37.74%	183	109	40.44%	188	109	42.02%
HardBrake	223	196	12.11%	217	183	15.67%	196	166	15.31%
Total Viol.	1368	1,026	25.00%	1258	831	33.94%	1225	826	32.57%

5.5.3 RQ2: Effectiveness at Producing Scenarios with Safety and Comfort Violations

RQ2 investigates whether (SCENORITA⁺⁺) leads to more reported violations, compared to random search (RANDOM) and a partially-mutable representation (SCENORITA⁻). In conducting our evaluation of this research question, we followed the guidelines in [54] for comparing SCENORITA⁺⁺ against both SCENORITA⁻ and RANDOM. Hence, we performed two statistical tests: 1) *Mann-Whitney U-test p-values* to determine statistical differences, and 2) *Vargha-Delaney’s \hat{A}_{12} index* [170] to determine the effect size. The results of these tests, in our experiments, are interpreted as follows: If Mann-Whitney U-test produces $p \leq 0.05$, this indicates that there is a significant difference between the quality of solutions provided by SCENORITA⁺⁺ and SCENORITA⁻ or RANDOM. The \hat{A}_{12} statistical test measures how

often, on average, one approach outperforms another; if $\hat{A}_{12} = 0.5$, then the two approaches achieve equal performance; if $\hat{A}_{12} > 0.5$, then the first approach is better; otherwise, the first approach is worse. The closer \hat{A}_{12} is to 0.5, the smaller the difference between the techniques; the further \hat{A}_{12} is from 0.5, the larger the difference.

Table 5.4 and Figure 5.9 summarizes the number of *unique* violations found, by each representation, for all maps. From Table 5.4, we observe that SCENORITA⁺⁺ found, on average, a total of 1,026 unique violations in all maps over the course of our experiments. Furthermore, we find that SCENORITA⁺⁺ discovered 23.47% more violations compared to SCENORITA⁻ ($\hat{A}_{12} = 0.84, p < 0.05$), and 24.21% more violations compared to RANDOM ($\hat{A}_{12} = 0.89, p < 0.05$).

Table 5.5: The Average number of all violations (All Viol.) reported by three testing techniques (Test Tech.): SCENORITA⁺⁺, SCENORITA⁻ and RANDOM on three maps (Borregas, San Mateo, and Sunnyvale), along with the average, minimum, and maximum number of unique violations (Unique Viol.), and the percentage of duplicate violations eliminated (Elim. (%)). We highlight cells with the best reported results in grey.

Map	Test Tech.	Total Violations				Collision				Speeding				Unsafe Lane Change				Fast Acceleration				Hard Braking									
		Unique Viol.			Elim. (%)	Unique Viol.			Elim. (%)	Unique Viol.			Elim. (%)	Unique Viol.			Elim. (%)	Unique Viol.			Elim. (%)										
		All Viol.	Avg.MinMax.			All Viol.	Avg.MinMax.			All Viol.	Avg.MinMax.			All Viol.	Avg.MinMax.			All Viol.	Avg.MinMax.												
		All Viol.	Avg.MinMax.		Elim. (%)	All Viol.	Avg.MinMax.		Elim. (%)	All Viol.	Avg.MinMax.		Elim. (%)	All Viol.	Avg.MinMax.		Elim. (%)	All Viol.	Avg.MinMax.		Elim. (%)										
Borregas	sceno-RITA ⁺⁺	524	363	336	411	30.73%	196	180	150	198	8.16%	0	0	0	0	0.00%	110	51	33	93	53.64%	137	61	41	90	55.47%	81	71	51	78	12.35%
	sceno-RITA ⁻⁻	520	296	254	390	43.08%	185	122	117	128	34.05%	0	0	0	0	0.00%	122	48	27	106	60.66%	124	57	47	84	54.03%	89	69	37	83	22.47%
	Random	508	318	287	345	37.40%	183	148	121	163	19.13%	0	0	0	0	0.00%	116	51	29	93	56.03%	129	53	42	63	58.91%	80	66	48	76	17.50%
San Mateo	sceno-RITA ⁺⁺	377	313	257	358	16.98%	120	113	98	126	5.83%	18	14	10	17	22.22%	125	84	60	123	32.80%	50	49	39	57	2.00%	64	53	43	67	17.19%
	sceno-RITA ⁻⁻	316	237	226	252	25.00%	85	69	57	79	18.82%	14	10	5	15	28.57%	122	76	64	98	37.50%	37	34	23	42	8.11%	58	48	40	62	17.24%
	Random	300	225	212	240	25.00%	73	68	63	78	6.85%	16	10	9	12	37.70%	123	72	48	81	41.46%	37	36	24	44	2.70%	51	39	33	43	23.53%
Sunnyvale	sceno-RITA ⁺⁺	467	350	319	380	25.05%	95	93	74	106	2.11%	7	7	5	12	0.00%	262	156	115	183	40.46%	25	22	17	30	12.00%	78	72	61	85	7.69%
	sceno-RITA ⁻⁻	422	298	227	357	29.38%	58	55	45	70	5.17%	10	8	7	11	20.00%	262	151	96	181	42.37%	22	18	14	24	18.18%	70	66	57	83	5.71%
	Random	417	283	253	304	32.13%	49	48	41	52	2.04%	11	8	5	13	27.27%	270	146	116	160	45.93%	22	20	15	24	9.09%	65	61	56	70	6.15%

For the collision violation, SCENORITA⁺⁺ finds, on average, 386 collisions: a 56.91% increase

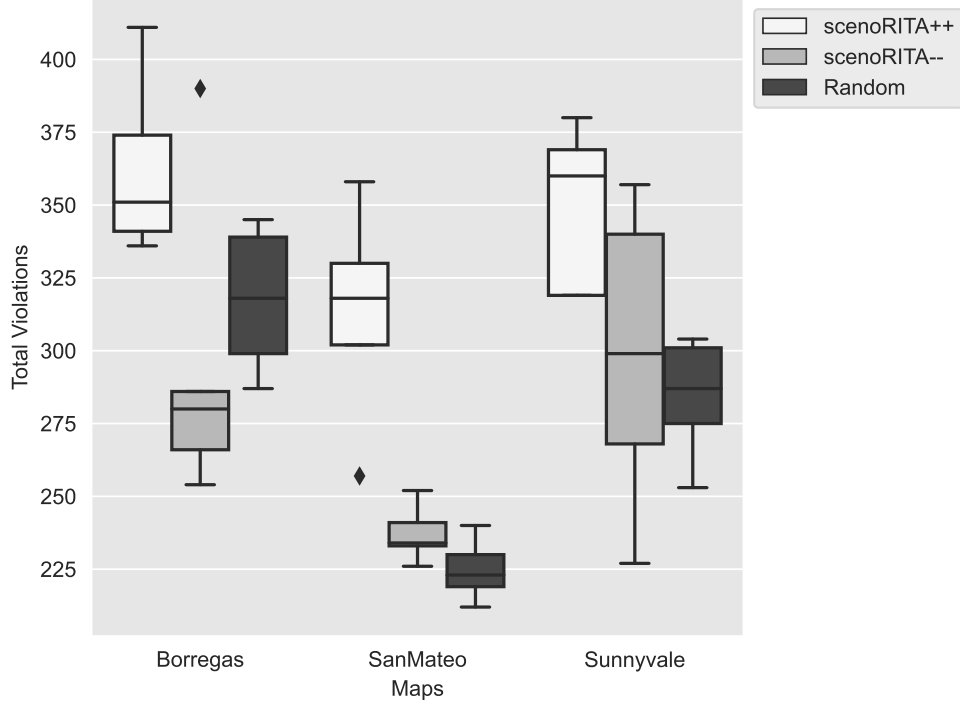


Figure 5.9: The total number of violations in tests reported by SCENORITA⁺⁺, SCENORITA⁻⁻, and RANDOM

compared to SCENORITA⁻⁻ ($\hat{A}_{12} = 0.80$), and a 46.21% increase compared to RANDOM ($\hat{A}_{12} = 0.77$). We observe a similar trend with the fast acceleration oracle, where SCENORITA⁺⁺ reports 21.10% more fast acceleration violations compared to each SCENORITA⁻⁻ and RANDOM, respectively ($\hat{A}_{12} = 0.63$ and $\hat{A}_{12} = 0.61$).

SCENORITA⁺⁺ reports a 16.67% increase in speeding violations, on average, compared to SCENORITA⁻⁻ and RANDOM. Furthermore, our results using the \hat{A}_{12} measure, indicate that SCENORITA⁺⁺ *statistically* outperforms the latter approaches: For 57% and 58% of the time, SCENORITA⁺⁺ reports more speed violations compared to RANDOM ($\hat{A}_{12} = 0.57$) and SCENORITA⁻⁻ ($\hat{A}_{12} = 0.58$). Similarly, SCENORITA⁺⁺ finds 13 more hard-braking violations compared to SCENORITA⁻⁻, and 30 more violations compared to RANDOM; SCENORITA⁺⁺ reports more hard-braking violations in 59% and 71% of the time compared to the other approaches. As for unsafe lane change, SCENORITA⁺⁺ finds, on average, 291 violations; a 5.82% increase compared to SCENORITA⁻⁻ ($\hat{A}_{12} = 0.54$), and a 8.18% compared to RANDOM

($\hat{A}_{12} = 0.54$).

The collision violations obtained with SCENORITA⁺⁺ were significantly higher than SCENORITA⁻ and RANDOM, compared to the other violations (i.e., speeding, unsafe lane change and hard braking). This result is likely due to the fact that the collision detection encodes obstacles’ behaviours in its fitness function (Section 5.4.2). One potential way to improve the reported violations for other oracles can be achieved by encoding more elements into the fitness function of other violations. For example, in the case of speed detection, we can encode—into its fitness function—the *number of times* an ego car was close to violating a lane’s speed limit. In order to add more complexity to the fitness evaluation of these violations, we need a better understanding of what causes a specific violation to occur more often (i.e., is it affected by more or less traffic, specific road curvatures, the existence or absence of traffic lights and stops signs, etc).

You may notice that there’s a small difference in the reported results between SCENORITA⁻ and RANDOM. We believe the reason can be attributed to the powerful nature of random search. As it has been shown in prior work [72], random exploration can achieve higher coverage compared to sophisticated strategies in other tools.

Table 5.5 shows an exhaustive list of unique violations associated with *each* map. From this table we observe that SCENORITA⁺⁺ found 47.54% and 21.62% more collisions, in Borregas Avenue, compared to SCENORITA⁻ and RANDOM, respectively. For San Mateo, SCENORITA⁺⁺ found 63.77% and 66.18% more collisions compared to SCENORITA⁻ and RANDOM; while in Sunnyvale, it found 69.09% and 93.75% more collision violations compared to the other techniques. Notice that the number of collision violations generated by each testing technique is *negatively correlated* with the size of the map. i.e., these techniques generate more violations in the smallest/simplest map (Borregas Ave) compared to the largest and most complex map (Sunnyvale). However, SCENORITA⁺⁺ performs *particularly* better in *Sunnyvale* compared to the other techniques (i.e., it generates 69.09% and 93.75% more

collisions compared to SCENORITA⁻ and RANDOM). We believe that the mutable gene representation in SCENORITA⁺⁺ enables it to outperform the other techniques, especially when running them on larger and more complex maps. As shown in Figure 5.7, Sunnyvale contains the largest number of lanes and the most complex and diverse roads (e.g., single- or multi-lane roads with either same or opposite traffic direction, U-turns, roundabouts, cross or T intersections, merged lanes, etc.). Borregas on the other hand, contains a maximum of two intersections with straight single-lane roads. No speeding violations were reported by any testing technique in Borregas Ave. We believe that the simplicity of driving scenarios generated in Borregas (i.e., the fact that, most of the time, the ego car drives on a single-lane straight road) prevented these techniques from finding speeding violations. Overall, we found that the number of speeding violations reported by any technique in any map, was significantly smaller compared to other violations. One reason might be that Apollo’s current implementation sets a very strict threshold for the maximum speed that an ego car can exceed, resulting in the ego car driving at a relatively slow speed most of the time.

Similarly, more unsafe lane change violations were reported, by these techniques, in Sunnyvale compared to Borregas. As mentioned earlier, Sunnyvale contains multi-lane roads with many turns and intersections that allows the ego car to perform more diverse driving maneuvers leading to more unsafe lane change violations. SCENORITA⁺⁺ generates more unsafe lane change violations compared to SCENORITA⁻ and RANDOM in each map. From these results, we find that:

Finding 2: In our experiments, SCENORITA⁺⁺ found a total of **1,026** safety and comfort violations including: **386** collisions, **21** speed violations, **291** unsafe lane changes, **132** fast acceleration violations, and **196** hard-braking violations. Overall, SCENORITA⁺⁺ finds, on average, **23.47%** more violations compared to SCENORITA⁻, and **24.21%** more violations compared to RANDOM.

Table 5.6 contains a list of ten case studies that demonstrate SCENORITA’s ability to generate effective and valid scenarios which expose the ego car to critical situations. These case studies demonstrate Apollo’s limited ability to handle unexpected behaviours by other obstacles (bicycles, pedestrians, and cars) such as speeding, not yielding to traffic, jaywalking, changing lanes suddenly, etc. This is alarming since (1) aggressive drivers, jaywalkers, and unexpected weather and road conditions are common in real life and (2) AV software, such as Apollo, should cope with such conditions. Similar problems have been revealed in Uber’s self-driving car that killed a jaywalker in Arizona in 2018 [26]. We discuss one case study in detail (case study 9) and refer the reader to our dataset [1] for video recordings and details of the remaining case studies.

Case Study. Figure 5.8 shows a scenario with two reported violations: **collision** and **hard braking**. The ego car is shown approaching an intersection at the beginning of the scenario (0 seconds). The ego car fails to predict the movement of another obstacle (a truck) crossing the same intersection from the left side. At 7 seconds, a decision to “yield to an object” (purple rectangle) is issued by the planning module, causing the ego car to brake suddenly and stop abruptly at 10 seconds; the speed of the ego car went down from 26 km/hr to 0 km/hr within less than 2 seconds, resulting in a motion sickness-inducing deceleration value of $-4.3 m/s^2$. Since the ego car stopped in the middle of an intersection with oncoming traffic, the truck travelling on its left collides with it at 11 seconds. These two violations occur due to the planning module being unable to react to a sudden change in an obstacle behaviour.

Initially, the prediction module detects that the obstacle travelling on the left would make a U-turn at the intersection, instead, the obstacle continues driving straight. As the ego car prepares to cross the intersection, it receives a new decision to yield to the obstacle driving on its left. Since the decision to yield to oncoming traffic came in late, the ego car was unable to handle this decision on time. As a result, the ego car stops abruptly in the middle of an

intersection with oncoming traffic, causing a hard brake followed by a collision violation.

From these case studies, we determine the following finding:

Finding 3: The case studies show that SCENORITA is capable of generating complex and effective scenarios that expose the ego car to critical and realistic situations. Two of the case studies demonstrate Apollo’s limited ability to cope with unexpected behaviours from obstacles, such as aggressive drivers or jaywalkers.

5.5.4 RQ3: Efficiency of scenoRITA

In RQ3, we study the efficiency of SCENORITA⁺⁺ by measuring its execution time, and comparing it with the execution time of SCENORITA⁻ and RANDOM. Table 5.7 shows that SCENORITA⁺⁺ takes **63.92** seconds, on average, to execute a scenario from end-to-end (*E2E*); it takes **10.43** seconds, on average, to generate the scenario representation and confirm its validity according to the *domain-specific constraints* (*MISC*); **41.52** seconds to generate the corresponding driving simulation (*Simulation*); and **11.97** seconds for checking the grading metrics (*Oracles*). Simulating driving scenarios is time-consuming (e.g., transforming the scenario representation into simulations, running each simulation for 30 seconds, and recording the car behaviour); hence, the scenario simulation stage strongly affects the efficiency of the overall test generation process in SCENORITA. We further observe that the difference in execution-time between all three representations (SCENORITA⁺⁺, SCENORITA⁻, and RANDOM) is negligible. From these results, we find that:

Finding 4: SCENORITA⁺⁺ is efficient, with an average runtime of 63.92 seconds per scenario, and can be used in practice to generate driving scenarios that expose AV software to safety violations. Moreover, SCENORITA⁺⁺ managed to generate 23.47% and 24.21% more violations compared to the two other representations in the same amount of time.

Table 5.6: Ten case studies with reported violations generated by SCENORITA, along with a description of the scenarios. The videos corresponding to these case studies can be found in [1].

Case Study No.	Map	Violations	Description
1	San Mateo	Collision	The ego car doesn't yield to oncoming traffic at an intersection; instead of allowing an obstacle (truck) to cross the intersection (the truck has the right-of-way), the ego car turns right and collides with the oncoming truck.
2	San Mateo	Speeding & Unsafe Lane Change	In an attempt to reach the final destination within the duration of the scenario (30 seconds), the ego car increases its speed (52 km/hr) until it exceeds the speed limit imposed by the lane its travelling on (40 km/hr). Moreover, the ego car drives on multiple lane boundaries for more than 5 seconds resulting in an unsafe lane-change violation.
3	San Mateo	Hard Braking	The ego car didn't detect a pedestrian jaywalking; the decision to yield to the pedestrian came late and resulted in the ego car braking too hard to avoid colliding with the pedestrian.
4	San Mateo	Fast Acceleration & Hard Braking	The ego car accelerates from an initial speed of 1 km/hr to 30 km/hr within 1.53 seconds (acceleration value of $5.28m/s^2$). A few seconds later, an obstacle (a truck)—driving in the next lane (opposite direction), passes by the ego car. The control modules receives a false "yield sign in-front" decision, causing the ego car to brake suddenly and too hard (deceleration value of $-8.2m/s^2$).
5	Borregas	Hard Braking	The ego car could not predict whether an obstacle (a bicycle) is driving straight or turning left at an intersection. As a result, a decision to "yield" to the obstacle is triggered, to prevent a collision. However, this decision came in late, causing the ego car to apply a hard brake.
6	Borregas	Collision & Fast Acceleration	The ego car didn't yield to an obstacle (a car) travelling at a high speed at an intersection. The ego car attempted to avoid the collision by accelerating too fast, but still couldn't avoid the collision.
7	Sunnyvale	Collision & Fast Acceleration	The ego car approaches an intersection and attempts to turn right, while another obstacle (a car) approaches the same intersection from the left side. The ego car mispredicts the trajectory of the obstacle; it predicted that the obstacle would make a left turn instead of driving straight, so it proceeds to turn right. The obstacle continues to drive straight and collides with the ego car.
8	Sunnyvale	Collision & Unsafe Lane Change	The ego car didn't yield to oncoming traffic at an intersection resulting in an accident. The ego car mispredicted the speed and distance of other obstacles, and attempted to cross the intersection before they arrive.
9	Sunnyvale	Collision & Hard Braking	As the ego car is about to cross an intersection, it receives a decision to yield to oncoming traffic from its left. The decision to yield to obstacles caused the ego car to suddenly brake in the middle of the intersection causing the oncoming traffic to collide with it.
10	Sunnyvale	Collision	The ego car mispredicts the behaviour of an obstacle (a bicycle), assuming the bicycle will yield to the ego car, since the former is making a right turn at an intersection. The bicycle did not yield to the ego car, and this unexpected behaviour was not handled properly by the ego car, resulting in the ego car hitting the bicycle.

Table 5.7: Efficiency of generated scenarios by SCENORITA⁺⁺, SCENORITA⁻⁻ and RANDOM

	Execution Time (sec.)			
	Simulation	Oracles	MISC	E2E
SCENORITA ⁺⁺	41.52	11.97	10.43	63.92
SCENORITA ⁻⁻	42.09	12.26	10.17	64.51
RANDOM	41.60	12.07	9.16	62.83

5.5.5 RQ4: Duplicate Violation Detection

This RQ investigates the extent to which SCENORITA eliminates similar violations, and compares the percentage of duplicate violations generated by all three representations (SCENORITA⁺⁺, SCENORITA⁻, and RANDOM). To answer this RQ, we configure DBSCAN [86] to cluster the scenarios with similar violations into the same group, based on a set of features as described in Section 5.4.6. We adopted the approach in [154] to automatically determine the optimal value for *epsilon*; *epsilon* defines the maximum distance allowed between two points within the same cluster. Eliminating duplicate violations is quick and takes, on average, 0.1 seconds per experiment.

To confirm the correctness of generated clusters, three of the authors manually and independently evaluated the accuracy of generated clusters in 18 randomly-selected experiments out of a total of 45 (40%). The authors examined violations in the same clusters to confirm whether they are similar by comparing a set of features associated with each scenario in the cluster. For example, consider two scenarios, **Scenario1** and **Scenario17**, both of which are in the same cluster and have a collision violation: In such a case, we compare the set of features (from Section 5.4.6) of the two scenarios to confirm that the collision occurred in the same position in **Scenario1** as it did in **Scenario17** ($p_{t_c}^E$), the ego car in both scenarios collided with an obstacle with the same type (T_{O_k}) and size (Z_{O_k}), the crash in both scenarios is the same ($collision^{type}$), etc. The authors also replayed these scenarios on Dreamview to observe if the scenarios in one cluster have similar violations.

Table 5.4 shows all violations (including duplicates) generated by SCENORITA⁺⁺, SCENORITA⁻, and RANDOM along with the unique number of violations (generated by the *Duplicate Violations Detector*), and the percentage of eliminated violations. From the results in Table 5.4, we observe that SCENORITA⁺⁺ eliminated, on average, a total of 342 similar violations in all maps over the course of our experiments. Furthermore, we observe that

SCENORITA⁺⁺ eliminated fewer violations (25%) compared to SCENORITA⁻ (33.94%) and RANDOM (32.57%)—indicating that SCENORITA⁺⁺, overall, generates effective driving scenarios with less redundant violations.

Similarly, SCENORITA⁺⁺ eliminated fewer collision violations (6.08%), followed by RANDOM (13.44%), and finally SCENORITA⁻ with 25% eliminated duplicates. We notice a similar trend with the remaining violations, where SCENORITA⁺⁺ generates more unique violations overall. Another interesting observation is that, certain violation types tend to have more duplicates compared to others. For instance, all three representations generate tests with more than 37% duplicate fast-acceleration violations, but with less similar hard-braking violations (12.11% - 15.67%).

Table 5.5 shows that the majority of eliminated fast acceleration violations are in Borregas; 55.47% eliminated in SCENORITA⁺⁺, 54.03% in SCENORITA⁻ and 58.91% in RANDOM. We believe that the size of the map (60 lanes with a total length of 3km) and the limited number of road curvature (Borregas contains only two intersections with traffic lights, and one road with two lanes without any stop signs or traffic lights) is the main reason why more similar fast-acceleration violations were found in Borregas compared to other maps.

Overall, we found that SCENORITA⁺⁺ eliminates fewer duplicate violations compared to other testing techniques. In the few cases where it eliminated more duplicate violations, SCENORITA⁺⁺ was still close enough to the winning technique. For example, in Sunnyvale, RANDOM eliminated 2.04% duplicate collisions while SCENORITA⁺⁺ eliminated 2.11%. However, SCENORITA⁺⁺ still reported almost twice as many collisions compared to RANDOM, due to fewer duplicate violations not necessarily resulting in more unique violations.

Finding 5: Our manual verification of 40% of our experiments shows that SCENORITA is able to identify and eliminate duplicate tests. The *Duplicate Violations Detector* eliminated 25% duplicate tests in SCENORITA⁺⁺, 33.94% in SCENORITA⁻, and 32.57% in RANDOM—

indicating that SCENORITA ⁺⁺ generates more unique violations compared to the two other representations.

5.6 Threats to Validity

Internal threats. One potential threat to internal validity is the selection of scenario duration: Simulation-based tests require the execution of time-consuming computer simulations to produce violations. We determined from our experimentation that our selected scenario duration of 30 seconds finds a significant number and variety of violations without incurring drastically long test execution times.

Another threat to validity is related to choosing DBSCAN (i.e., density-based spatial clustering of applications with noise) in eliminating duplicate violations. To mitigate this threat to validity, we ran both k-means and DBSCAN on a random set of scenarios with duplicate violations, then manually inspected the clusters of scenarios with similar violations generated by both techniques. We found that (i) k-means resulted in clusters of undesired structure and quality; (ii) unlike DBSCAN, k-means is not an ideal algorithm for latitude-longitude spatial data because it minimizes variance, not geodetic distances; (iii) DBSCAN is deterministic compared to k-means; and (iv) DBSCAN does not require to specify the number of clusters in advance—it determines them automatically based on epsilon, where epsilon defines the maximum distance allowed between two points within the same cluster. We avoided using hierarchical clustering due to its computationally expensive nature.

Another threat to validity arises from verifying the correctness of generated tests. There is unfortunately no automated strategies nor a ground truth that can be used to, otherwise, assess the accuracy of generated scenarios. For that reason, we had to manually verify the generated tests. We want to enable other researchers and practitioners to compare and verify the correctness of their tests by using the driving scenarios generated by SCENORITA as a

ground truth.

To account for validity threats arising from randomness in search algorithms, we follow the guidelines in [54]: (i) we repeated the experiments for each representation (SCENORITA⁺⁺, SCENORITA⁻, and RANDOM) 15 times, (ii) we used the non-parametric Mann-Whitney U-test to detect statistical differences and reported the obtained p -value, and (iii) we reported \hat{A}_{12} index (a standardized effect size measure). We make our full experimental results available in [1].

To mitigate threats arising from our selection of search operators, we selected (i) a widely-used algorithm in the search-based software engineering (SBSE) community, i.e., NSGA-II and (ii) crossover and mutation algorithms that best fits our gene representation. For parameter tuning, we followed the guidelines in [55, 54]—which suggests that standard parameter settings are usually recommended—leading us to use default settings in DEAP-1.3 [94], the framework used in our search-based implementation.

External threats. One external threat is that we applied SCENORITA to a single AV software system, Apollo. To mitigate the threat, we selected the only high autonomy (i.e., Level 4), open-source, production-grade AV software system that supports a wide variety of driving scenarios and explicitly aims for both safety and driver comfort. To mitigate threats related to generalizability of our results to other maps, we applied SCENORITA to three high-definition maps of cities in California: Borregas (60 lanes), San Mateo (1,305 lanes), and Sunnyvale (3,061 lanes). Note that Autoware [13], despite being open-source and widely-used [35], is considered a research-grade and not a production-grade AV software system [117, 118], which we further verified through speaking with Christian John, the Vice Chair and Chief Software Architect of Autoware.

Construct Validity. The main threat to construct validity is how we measure and calculate safety and comfort violations. To mitigate this threat, we measure these violations using

grading metrics defined by Apollo’s developers [19]. We utilize thresholds (e.g., speeding or acceleration thresholds) set by Apollo’s developers [19]; the U. S. Department of Transportation [92]; or thresholds used by major AV companies (e.g., Alphabet Waymo [22]).

5.7 Discussion

In this chapter, we propose SCENORITA, a novel search-based testing framework, which exposes AV software to 3 types of safety-critical and 2 types of motion sickness-inducing scenarios in a manner that reduces duplicate scenarios, allows fully mutable obstacles with valid and modifiable obstacles trajectories, and follows domain-specific constraints obtained from authoritative sources. We evaluate SCENORITA on Baidu Apollo, a high autonomy (L4), open-source, and production-grade AV software system that supports a wide variety of driving scenarios. We compare our approach (SCENORITA⁺⁺) with a state-of-the-art search-based testing approach using only a partially mutable representation (SCENORITA⁻) and a random version of our approach that leverages SCENORITA’s *domain-specific constraints* and randomly-generated obstacles. SCENORITA⁺⁺ found a total of 1,026 unique safety and comfort violations including: 386 collisions, 21 speed violations, 291 unsafe lane changes, 132 fast acceleration violations, and 196 hard-braking violations. Moreover, SCENORITA⁺⁺ generates, on average, 23.47% and 24.21% more violations compared to the two other representations in the same amount of time (63.92 sec/scenario). For future work, we aim to expand SCENORITA to handle (i) generation of scenarios and oracles focused on traffic lights and stop signs and (ii) extending the work to other AV software systems (e.g., Autoware).

Chapter 6

Conclusion

Smart systems are software entities that carry out a set of operations on behalf of a user or another application with some degree of independence or autonomy. These systems employ some knowledge or representations of (1) a user's goals or desires, and (2) the environment in which they act in to achieve these goals. These systems address environmental, societal, and economic challenges like limited resources, climate change, and globalization. They are, for that reason, increasingly used in a large number of sectors such as transportation, healthcare, energy, safety, security, etc. Hence, the need for effective analysis and testing techniques for such systems has increased more than ever.

In this thesis, I propose to ensure software dependability, specifically software security, safety, and reliability of smart systems. In particular, I conduct comprehensive studies of bugs and vulnerabilities found in such systems, then follow these studies with tools and strategies that enable us to automatically test and detect vulnerabilities and bugs found in smart systems. Section 6.1 lists the research contributions of this thesis in details.

6.1 Research Contributions:

This dissertation makes the following contributions to the Software Engineering research community:

- **Automatic identification of native libraries' versions in Android apps.** I constructed a novel approach (*LibRARIAN*) that, given an unknown binary, identifies (i) the library it implements and (ii) its version. Furthermore, I introduced a new similarity-scoring mechanism for comparing native binaries which utilizes 6 features that enable *LibRARIAN* to distinguish between different libraries and their versions. In our ground truth dataset which contains 46 known libraries with 904 versions, *LibRARIAN* correctly identifies 91.15% of those library versions, thus achieving a high identification accuracy. *LibRARIAN* also achieves a 12% improvement in its accuracy compared to *OSSPolice*, the state-of-the-art technique for identifying versions of native binaries for Android apps.
- **Longitudinal study of security updates in Android Apps' native code.** I conducted a large-scale, longitudinal study that tracks security vulnerabilities in native libraries used in Android apps over 7 years. In particular, I utilized *LibRARIAN* to study (1) the prevalence of vulnerabilities in native libraries in the top 200 apps, and (2) the rate at which app developers apply security patches.
- **A comprehensive study of autonomous vehicle bugs.** We conducted the first comprehensive study of bugs in AV systems through a manual analysis of 499 bugs obtained from 16,851 commits in two dominant AV open-source software systems. In this study, we provided a classification of root causes, symptoms, and the AV components affected by these bugs. Based on this study, we discussed and suggested future research directions related to software testing and analysis of AV systems.

- **Automatic generation of diverse, fully-mutable, safety-critical, and motion sickness-inducing scenarios for Autonomous Vehicles.** I implemented a search-based testing framework, with a novel gene representation and domain-specific constraints, that automatically generates valid, fully mutable, and diverse driving scenarios. To improve the effectiveness of this approach, we automate the process of identifying and eliminating duplicate violations by using an unsupervised clustering technique to group driving scenarios, with similar violations, according to specific features. Furthermore, we utilize 5 test oracles and corresponding fitness functions to assess different aspects of AVs—ranging from traffic and road safety (i.e., collision detection, speeding detection, and unsafe lane change) to a rider’s comfort (i.e., fast acceleration and hard braking).

- **Tools and experiments:**

- *Automatic identification of native libraries’ versions in Android apps.* I implemented the proposed technique and evaluated it on 46 known libraries with 904 versions. This tool is publicly available [134] to enable reusability, reproducibility, and others to build upon our work.
- *Automatic generation of diverse, fully mutable, safety-critical, and motion sickness-inducing scenarios for autonomous vehicles.* I implemented the proposed technique and evaluated it on Baidu Apollo using 3 high-definition maps of cities/street blocks located in California. This tool is publicly available [1] to enable reusability, reproducibility, and others to build upon our work.

- **Datasets:**

- *A dataset of 66,684 native third-party libraries in the top 200 popular Android Apps.* I built a repository of Android apps and their native libraries with the 200 most popular free apps from Google Play totaling 7,678 versions gathered between the dates of Sept. 2013 and May 2020. This repository further contains 66,684 native libraries used by these 7,678 versions. Moreover, we built a ground truth

dataset which contains 46 known third-party libraries (such as FFmpeg, GIFLib, OpenSSL, WebP, SQLite3, OpenCV, Jpeg-turbo, Libpng, and XML2) and their 904 library versions. We make our dataset, analysis platform, and results available online to enable reusability, reproducibility, and others to build upon our work [134].

- *A dataset of 499 bugs obtained from 16,851 commits in two dominant AV open-source software systems.* We further provide details of the root causes associated with each of these bugs, their symptoms and the AV component affected by them. 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[6].

6.2 Future Work:

- **Software Security in Smartphones:** Possible follow-up work of my research on security updates in Android apps’ native code include: (1) extending *LibRARIAN* by using machine learning to improve the identification threshold for determining versions or library names and automatically extracting heuristics for Version Identification Strings. This extension aims to improve the current accuracy of *LibRARIAN* and to reduce some of the manual labor in extracting version information from libraries; (2) conducting user interviews to get a better understanding of why app developers are slow in terms of updating their libraries to patched versions. This study aims to provide a thorough discussion of solutions and actionable items for different actors in the app ecosystem to remedy this situation effectively; (3) building a cross-language control-flow/data-flow analysis to assess the reachability of vulnerable native code from the Dalvik code of an Android app which will further enable (4) building an automatic exploit generation of vulnerable libraries; and (5) implementing mechanisms to automatically update native

libraries while also testing for regressions and possibly automatically repairing them, or, at least, notifying developers about vulnerable libraries. Such an idea is similar to how Debian’s repositories centrally manage libraries/dependencies between applications.

- **Software Reliability and Safety in Smart Cars:** Interesting future work based on our AV bug study and SCENORITA include: (1) extending SCENORITA to find more violations and violation types by incorporating additional oracles and generating complex driving scenarios taking into account other attributes such as traffic lights, crosswalks, and stop signs; (2) generating tests with bug-revealing violations; (3) designing an “ideal” framework to find all possible violations in a driving simulation. This NP-hard, optimal framework requires moving an AV simulator (ex. Apollo Simulator) to a quantum computer to enumerate all possible scenarios; (4) conducting a comprehensive study to understand the fix patterns of bugs found in AV software and to identify the challenges associated with fixing such bugs. This study will inform strategies for repairing AV software.
- **Software Reliability in Large-Scale Cloud Services:** Production incidents in today’s large-scale cloud services can be extremely expensive in terms of customer impacts and engineering resources required to mitigate them. Despite continuous reliability efforts, cloud services still experience severe incidents due to various root causes. Many of these incidents last for a long period as existing techniques and practices fail to detect and mitigate them quickly. To better understand these issues, an in-depth study of high severity production incidents is needed. This study should answer the following questions: (1) what bugs are behind these incidents? (2) when and how were these bugs introduced? (3) why did these bugs escape testing? (4) why were they not caught in monitoring? (5) why were they not mitigated quickly? (6) why are existing fault tolerance tools not effective? and (7) how can we prevent these incidents from happening again? This study will provide data-driven recommendations

to improve the effectiveness of current tools, and suggest new tools and extensions with the goal of improving cloud reliability.

Bibliography

- [1] scenorita: Open-source framework for generating safety critical scenarios for avs, Aug 2021.
- [2] 46 Corporations Working On Autonomous Vehicles. <https://www.cbinsights.com/research/autonomous-driverless-vehicles-corporations-list/>, August 2019.
- [3] Apollo Cyber RT. <https://github.com/ApolloAuto/apollo/tree/master/cyber>, August 2019.
- [4] Bazel. <https://bazel.build/>, August 2019.
- [5] CMake. <https://cmake.org/>, August 2019.
- [6] A comprehensive study of autonomous vehicle bugs - artifact website. http://tiny.cc/cps_bug_analysis, August 2019.
- [7] Github: The CARMA platform. <https://github.com/usdot-fhwa-stol/CARMAPlatform>, August 2019.
- [8] Self-Driving Uber Car Kills Pedestrian in Arizona, Where Robots Roam. <https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html>, August 2019.
- [9] The 18 Companies Most Likely to Get Self-driving Cars on the Road First. <https://www.businessinsider.com/the-companies-most-likely-to-get-driverless-cars-on-the-road-first-2017-4>, August 2019.
- [10] USDOT: The CARMA platform. <https://highways.dot.gov/research/research-programs/operations/CARMA>, August 2019.
- [11] Waymo Launches Self-driving car Service Waymo One. <https://techcrunch.com/2018/12/05/waymo-launches-self-driving-car-service-waymo-one>, August 2019.
- [12] *Apollo Dreampview*, August 2021.
- [13] Autoware: Open-source software for urban autonomous driving, August 2021.

- [14] Avoiding carsickness when the cars drive themselves, August 2021.
- [15] Baidu apollo: An open autonomous driving platform, August 2021.
- [16] Baidu debuts robotaxi ride hailing service in china, using self-driving electric taxis, August 2021.
- [17] Baidu hits the gas on autonomous vehicles with volvo and ford deals, August 2021.
- [18] Baidu starts mass production of autonomous buses, August 2021.
- [19] Dreamland's grading system, August 2021.
- [20] Ford unveils new self-driving test vehicle for 2022 launch, August 2021.
- [21] Google's self-driving car caused its first accident, August 2021.
- [22] Look, no hands! test driving a google car, August 2021.
- [23] Measuring motion sickness in driverless cars, August 2021.
- [24] Open vehicles compatible with apollo, August 2021.
- [25] Self-driving tesla was involved in fatal crash, u.s. says, August 2021.
- [26] Self-driving uber car kills pedestrian in arizona, where robots roam, August 2021.
- [27] Tesla autopilot, August 2021.
- [28] Tesla autopilot system found probably at fault in 2018 crash, August 2021.
- [29] Tesla: Autopilot was on during deadly mountain view crash, August 2021.
- [30] Tesla driver dies in first fatal crash while using autopilot mode, August 2021.
- [31] There are some scary similarities between tesla's deadly crashes linked to autopilot, August 2021.
- [32] Two years on, a father is still fighting tesla over autopilot and his son's fatal crash, August 2021.
- [33] Uber advanced technology group, August 2021.
- [34] Udacity: Self-driving fundamentals: Featuring apollo, August 2021.
- [35] Vision and mission of autoware, August 2021.
- [36] Waymo, August 2021.
- [37] Waymo and intel collaborate on self-driving car technology, August 2021.
- [38] Waymo's autonomous cars have driven 8 million miles on public roads, August 2021.

- [39] You can take a ride in a self-driving lyft during ces, August 2021.
- [40] Autonomous vehicle startup autox lands driverless testing permit in california, Feb 2022.
- [41] 2021 IEEE International Conference On Artificial Intelligence Testing (AITest), July 2022.
- [42] Apollo’s Sim Control, July 2022.
- [43] Study: The most dangerous intersections in Orange county, July 2022.
- [44] The 14th Intl. Workshop on Search-Based Software Testing, July 2022.
- [45] The 15th Intl. Workshop on Search-Based Software Testing, July 2022.
- [46] U.S. agency to review if Pony.ai complied with crash reporting order, July 2022.
- [47] Raja Ben Abdesslem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1016–1026. IEEE, 2018.
- [48] Azat Abdullin, Marat Akhin, and Mikhail Belyaev. Kex at the 2021 sbst tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 32–33, 2021.
- [49] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. DroidNative: Automating and optimizing detection of android native code malware variants. *Computers & Security*, 65:230 – 246, 2017.
- [50] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR ’16*, pages 468–471. ACM Press, 2016.
- [51] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. Too quiet in the library: An empirical study of security updates in android apps’ native code. In *Proceedings of the 43rd International Conference on Software Engineering, ICSE ’21*, page 1347–1359. IEEE Press, 2021.
- [52] Matthias Althoff and Sebastian Lutz. Automatic generation of safety-critical test scenarios for collision avoidance of road vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1326–1333, 2018.
- [53] Gábor Antal, Márton Keleti, and Péter Hegedundefineds. Exploring the security awareness of the python and javascript open source communities. In *Proceedings of the 17th International Conference on Mining Software Repositories, MSR ’20*, page 16–20, New York, NY, USA, 2020. Association for Computing Machinery.

- [54] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, page 1–10, New York, NY, USA, 2011.
- [55] Andrea Arcuri and Gordon Fraser. On parameter tuning in search based software engineering. In *Search Based Software Engineering*, pages 33–47, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [56] Karl J Åström and Björn Wittenmark. *Adaptive control*. Courier Corporation, 2013.
- [57] Automated Vehicle Standards Committee”. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, June 2018.
- [58] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS’16*, pages 356–367. ACM Press, 2016.
- [59] Hanna Bellem, Barbara Thiel, Michael Schrauf, and Josef F Krems. Comfort in automated driving: An analysis of preferences for different automated driving styles and their dependence on personality traits. *Transportation research part F: traffic psychology and behaviour*, 55:90–100, 2018.
- [60] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 63–74, 2016.
- [61] Michele Bertoncello and Dominik Wee. Ten ways autonomous driving could redefine the automotive world. *McKinsey & Company*, 6, 2015.
- [62] Peter Biber and Wolfgang Straßer. 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)*, volume 3, pages 2743–2748. IEEE, 2003.
- [63] Gary Bishop, Greg Welch, et al. An introduction to the kalman filter. *Proc of SIGGRAPH, Course*, 8(27599-23175):41, 2001.
- [64] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal analysis of android ad library permissions. *arXiv:1303.0857 [cs]*, 2013.
- [65] Tomas Bures, Danny Weyns, Bradley Schmer, Eduardo Tovar, Eric Boden, Thomas Gabor, Ilias Gerostathopoulos, Pragya Gupta, Eunsuk Kang, Alessia Knauss, Pankesh Patel, Awais Rashid, Ivan Ruchkin, Roykrong Sukkerd, and Christos Tsigkanos. Software engineering for smart cyber-physical systems: Challenges and promising solutions. *SIGSOFT Softw. Eng. Notes*, 42(2):19–24, jun 2017.

- [66] N. Cacho, E. A. Barbosa, J. Araujo, F. Pranto, A. Garcia, T. Cesar, E. Soares, A. Cassio, T. Filipe, and I. Garcia. How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 31–40, September 2014.
- [67] Alessandro Calò, Paolo Arcaini, Shaukat Ali, Florian Hauer, and Fuyuki Ishikawa. Generating avoidable collision scenarios for testing autonomous driving systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 375–386, 2020.
- [68] Yulong Cao, Chaowei Xiao, Dawei Yang, Jing Fang, Ruigang Yang, Mingyan Liu, and Bo Li. Adversarial objects against lidar-based autonomous driving systems. *arXiv preprint arXiv:1907.05418*, 2019.
- [69] Brandon Carlson, Kevin Leach, Darko Marinov, Meiyappan Nagappan, and Atul Prakash. Open source vulnerability notification. In Francis Bordeleau, Alberto Sillitti, Paulo Meirelles, and Valentina Lenarduzzi, editors, *Open Source Systems - 15th IFIP WG 2.13 International Conference, OSS 2019, Proceedings*, IFIP Advances in Information and Communication Technology, pages 12–23. Springer New York LLC, January 2019. 15th International Conference on Open Source Systems, OSS 2019 ; Conference date: 26-05-2019 Through 27-05-2019.
- [70] Ezequiel Castellano, Ahmet Cetinkaya, Cédric Ho Thanh, Stefan Klikovits, Xiaoyi Zhang, and Paolo Arcaini. Frenetic at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 36–37, 2021.
- [71] Centers for Disease Control and Prevention, National Center for Health Statistics. *Anthropometric Reference Data for Children and Adults: United States, 2015–2018*, Jan 2021.
- [72] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE, 2015.
- [73] R. Coelho, L. Almeida, G. Gousios, and A. v Deursen. 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*, pages 134–145, May 2015.
- [74] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.
- [75] Anthony Corso, Peter Du, Katherine Driggs-Campbell, and Mykel J Kochenderfer. Adaptive stress testing with reward augmentation for autonomous vehicle validation. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 163–168, 2019.
- [76] Cve. <https://cve.mitre.org/>.

- [77] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000.
- [78] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18*, page 181–191, New York, NY, USA, 2018. Association for Computing Machinery.
- [79] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, pages 2187–2200. ACM Press, 2017.
- [80] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. A Comprehensive Study of Real-world Numerical Bug Characteristics. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 509–519, Piscataway, NJ, USA, 2017. IEEE Press. event-place: Urbana-Champaign, IL, USA.
- [81] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, pages 2169–2185. ACM Press, 2017.
- [82] H. Ebadi, M. Moghadam, M. Borg, G. Gay, A. Fontes, and K. Socha. Efficient and effective generation of test cases for pedestrian detection - search-based software testing of baidu apollo in svl. In *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 103–110, Los Alamitos, CA, USA, aug 2021. IEEE Computer Society.
- [83] William Enck, Damien Ocateau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *in Proceedings of the 20th USENIX Security Symposium*, page 16, 2011.
- [84] Christian Erbsmehl. Simulation of real crashes as a method for estimating the potential benefits of advanced safety technologies. In *21st International Technical Conference on the Enhanced Safety of Vehicles (ESV) National Highway Traffic Safety*, pages 09–0162, 2009.
- [85] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.
- [86] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

- [87] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramèr, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Physical Adversarial Examples for Object Detectors. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.
- [88] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning models. *arXiv preprint arXiv:1707.08945*, 2017.
- [89] F-droid. <https://f-droid.org>.
- [90] Federal Highway Administration, US Department of Transportation. *Federal Size Regulations For Commercial Motor Vehicles*, June 1992.
- [91] Federal Highway Administration, US Department of Transportation. *Bicycle Road Safety Audit Guidelines and Prompt Lists*, May 2012.
- [92] Federal Highway Administration, US Department of Transportation. *Analysis of Lane-Change Crashes and Near-Crashes*, June 2009.
- [93] Ffmpeg. <https://ffmpeg.org/>.
- [94] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175, 2012.
- [95] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.
- [96] Alessio Gambi, Tri Huynh, and Gordon Fraser. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 257–267, 2019.
- [97] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 318–328, 2019.
- [98] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jianguang Sun. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 896–899. ACM Press, 2018.
- [99] Carlos E Garcia, David M Prett, and Manfred Morari. Model predictive control: theory and practice—a survey. *Automatica*, 25(3):335–348, 1989.

- [100] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 385–396, New York, NY, USA, 2020. Association for Computing Machinery.
- [101] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 19–30. ACM, 2019.
- [102] gnu.org. <https://developer.android.com/ndk>.
- [103] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks - WISEC '12*, page 101. ACM Press, 2012.
- [104] Fitash Ul Haq, Donghwan Shin, and Lionel Briand. Efficient online testing for dnn-enabled systems using surrogate-assisted and many-objective optimization. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 811–822, 2022.
- [105] Florian Hauer, Ilias Gerostathopoulos, Tabea Schmidt, and Alexander Pretschner. Clustering traffic scenarios using mental models as little as possible. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 1007–1012, 2020.
- [106] Florian Hauer, Alexander Pretschner, and Bernd Holzmüller. Fitness functions for testing automated and autonomous driving systems. In *International Conference on Computer Safety, Reliability, and Security*, pages 69–84, 2019.
- [107] Qiang He, Bo Li, Feifei Chen, John Grundy, Xin Xia, and Yun Yang. Diversified Third-party Library Prediction for Mobile App Development. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.
- [108] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, page 63. ACM Press, 2011.
- [109] Dirk Holz, Alexandru E Ichim, Federico Tombari, Radu B Rusu, and Sven Behnke. Registration with the point cloud library: A modular framework for aligning in 3-d. *IEEE Robotics & Automation Magazine*, 22(4):110–124, 2015.
- [110] Hewlett packard enterprise cyber risk report 2016. https://www.thehaguesecuritydelta.com/media/com_hsd/report/57/document/4aa6-3786enw.pdf.
- [111] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. BinMatch: A semantics-based hybrid approach on binary code clone analysis. *arXiv:1808.06216 [cs]*, 2018-08-19.

- [112] Rasheed Hussain and Sherali Zeadally. Autonomous cars: Research results, issues, and future challenges. *IEEE Communications Surveys & Tutorials*, 21(2):1275–1313, 2018.
- [113] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A Comprehensive Study on Deep Learning Bug Characteristics. *arXiv:1906.01388 [cs]*, June 2019. arXiv: 1906.01388.
- [114] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’12, pages 77–88, New York, NY, USA, 2012. ACM. event-place: Beijing, China.
- [115] Stephen C Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.
- [116] Nidhi Kalra and Susan M Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 2016.
- [117] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.
- [118] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.
- [119] D. Kaufmann, L. Klampfl, F. Kluck, M. Zimmermann, and J. Tao. Critical and challenging scenario generation based on automatic action behavior sequence optimization: 2021 iee autonomous driving ai test challenge group 108. In *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 118–127, Los Alamitos, CA, USA, aug 2021. IEEE Computer Society.
- [120] Siddhartha Khastgir, Stewart Birrell, Gunwant Dhadyalla, and Paul Jennings. Identifying a gap in existing validation methodologies for intelligent automotive systems: Introducing the 3xd simulator. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 648–653. IEEE, 2015.
- [121] Jinhan Kim, Jeongil Ju, Robert Feldt, and Shin Yoo. Reducing dnn labelling cost using surprise adequacy: an industrial case study for autonomous driving. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1466–1476, 2020.
- [122] Florian Kluck, Martin Zimmermann, Franz Wotawa, and Mihai Nica. Genetic algorithm-based test parameter optimization for adas system testing. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 418–425, 2019.

- [123] Florian Klück, Lorenz Klampfl, and Franz Wotawa. Gabezier at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 38–39, 2021.
- [124] Philip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016.
- [125] Friedrich Kruber, Jonas Wurst, and Michael Botsch. An unsupervised random forest clustering technique for automatic traffic scenario categorization. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 2811–2818. IEEE, 2018.
- [126] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 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*, pages 517–530, New York, NY, USA, 2016. ACM. event-place: Atlanta, Georgia, USA.
- [127] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020.
- [128] Alexander Kc Leung and Kam Lun Hon. Motion sickness: an overview. *Drugs in context*, 8, 2019.
- [129] Robert V Levine and Ara Norenzayan. The pace of life in 31 countries. *Journal of cross-cultural psychology*, 30(2):178–205, 1999.
- [130] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael Sullivan, Siva Kumar Sastri Hari, Zbigniew Kalbarczyk, and Ravishankar Iyer. Av-fuzzer: Finding safety violations in autonomous driving systems. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 25–36. IEEE, 2020.
- [131] Li Li, Wu-Ling Huang, Yuehu Liu, Nan-Ning Zheng, and Fei-Yue Wang. Intelligence testing for autonomous vehicles: A new approach. *IEEE Transactions on Intelligent Vehicles*, 1(2):158–166, 2016.
- [132] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. LibD: Scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346, 2017.
- [133] Yibin Liao, Ruoyan Cai, Guodong Zhu, Yue Yin, and Kang Li. MobileFindr: Function similarity identification for reversing mobile binaries. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, volume 11098, pages 66–83. Springer International Publishing, 2018.
- [134] Librarian. <https://github.com/salmanee/Librarian>.
- [135] libvpx. <https://github.com/webmproject/libvpx/>.

- [136] Qin Lin, Wenshuo Wang, Yihuan Zhang, and John M. Dolan. Measuring similarity of interactive driving behaviors using matrix profile. In *2020 American Control Conference (ACC)*, pages 3965–3970, 2020.
- [137] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '15*, pages 89–103. ACM Press, 2015.
- [138] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 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*, pages 329–339, New York, NY, USA, 2008. ACM. event-place: Seattle, WA, USA.
- [139] Yixing Luo, Xiao-Yi Zhang, Paolo Arcaini, Zhi Jin, Haiyan Zhao, Fuyuki Ishikawa, Rongxin Wu, and Tao Xie. Targeting requirements violations of autonomous driving systems by dynamic evolutionary search. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–291, 2021.
- [140] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. LibRadar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, pages 653–656. ACM Press, 2016.
- [141] Alejandro Mazuera-Rozo, Jairo Bautista-Mora, Mario Linares-Vásquez, Sandra Rueda, and Gabriele Bavota. The android os stack and its vulnerabilities: an empirical study. *Empirical Software Engineering*, 24(4):2056–2101, 08 2019. Copyright - Empirical Software Engineering is a copyright of Springer, (2019). All Rights Reserved; Last updated - 2019-07-25.
- [142] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.
- [143] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *USENIX Security Symposium*, 2017.
- [144] Mahshid Helali Moghadam, Markus Borg, and Seyed Jalaleddin Mousavirad. Deeper at the sbst 2021 tool competition: Adas testing using multi-objective search. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 40–41, 2021.
- [145] Israel J. Mojica Ruiz, Meiyappan Nagappan, Bram Adams, Thorsten Berger, Steffen Dienst, and Ahmed E. Hassan. Analyzing Ad Library Updates in Android Apps. *IEEE Software*, 33(2):74–80, March 2016. Conference Name: IEEE Software.

- [146] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering*, 23(5):2901–2947, Oct 2018.
- [147] A. Narayanan, L. Chen, and C. K. Chan. AdDetect: Automated detection of android ad libraries using semantic analysis. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6, 2017.
- [148] V. Nguyen, S. Huber, and A. Gambi. Salvo: Automated generation of diversified tests for self-driving cars from existing maps. In *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 128–135, Los Alamitos, CA, USA, aug 2021. IEEE Computer Society.
- [149] Nvidia. Cuda programming guide, 2010.
- [150] Opus. <https://opus-codec.org/>.
- [151] Zi Peng, Jinqu Yang, Tse-Hsun Chen, and Lei Ma. A first look at the integration of machine learning models in complex autonomous driving systems: a case study on apollo. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1240–1250, 2020.
- [152] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
- [153] Radare2. <https://rada.re/n>.
- [154] Nadia Rahmah and Imas Sukaesih Sitanggang. Determination of optimal epsilon (eps) value on dbscan algorithm to clustering data on peatland hotspots in sumatra. In *IOP conference series: earth and environmental science*, volume 31, page 012012. IOP Publishing, 2016.
- [155] Ignacio Manuel Lebrero Rial and Juan P. Galeotti. Evosuites at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 30–31, 2021.
- [156] Marcelo Romero, Wided Guédria, Hervé Panetto, and Béatrix Barafort. Towards a characterisation of smart systems: A systematic literature review. *Computers in industry*, 120:103224, 2020.
- [157] SAE On-Road Automated Vehicle Standards Committee and others. Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems. *SAE Standard J*, 3016:1–16, 2014.

- [158] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. 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*, pages 149–157, New York, NY, USA, 2008. ACM.
- [159] Marija Selakovic and Michael Pradel. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 61–72, New York, NY, USA, 2016. ACM. event-place: Austin, Texas.
- [160] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FLEXDROID: Enforcing in-app privilege separation in android. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.
- [161] J. Seymour, D. Ho, and Q. Luu. An empirical testing of autonomous vehicle simulator system for urban driving. In *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 111–117, Los Alamitos, CA, USA, aug 2021. IEEE Computer Society.
- [162] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2017.
- [163] Sonatype - 2019 state of the software supply chain. <https://www.sonatype.com/2019ssc>.
- [164] Speex. <https://www.speex.org/>.
- [165] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. Misbehaviour prediction for autonomous driving systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 359–371, 2020.
- [166] Mengtao Sun and Gang Tan. NativeGuard: protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks - WiSec '14*, pages 165–176. ACM Press, 2014.
- [167] F. Thung, S. Wang, D. Lo, and L. Jiang. An Empirical Study of Bugs in Machine Learning Systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280, November 2012.
- [168] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *International Conference on Software Engineering (ICSE)*, 2018.

- [169] Yuchi Tian and Baishakhi Ray. 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*, pages 752–762, New York, NY, USA, 2017. ACM. event-place: Paderborn, Germany.
- [170] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
- [171] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.
- [172] K. Viswanadha, F. Indaheng, J. Wong, E. Kim, E. Kalvan, Y. Pant, D. J. Fremont, and S. A. Seshia. Addressing the iee av test challenge with scenic and verifai. In *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 136–142, Los Alamitos, CA, USA, aug 2021. IEEE Computer Society.
- [173] Vorbis. <https://xiph.org/vorbis/>.
- [174] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. Bug characteristics in blockchain systems: a large-scale empirical study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 413–424. IEEE, 2017.
- [175] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. WuKong: a scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, pages 71–82. ACM Press, 2015.
- [176] Wenshuo Wang, Aditya Ramesh, Jiacheng Zhu, Jie Li, and Ding Zhao. Clustering of driving encounter scenarios using connected vehicle trajectories. *IEEE Transactions on Intelligent Vehicles*, 5(3):485–496, 2020.
- [177] X. Xia, L. Bao, D. Lo, and S. Li. “automated debugging considered harmful” 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)*, pages 267–278, Oct 2016.
- [178] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–387. ACM, July 2020.
- [179] Huan Yu, Xin Xia, Xiaoqiong Zhao, and Weiwei Qiu. Combining Collaborative Filtering and Topic Modeling for More Accurate Android Mobile App Library Recommendation. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware - Internetware’17*, pages 1–6, Shanghai, China, 2017. ACM Press.

- [180] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deep-road: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 132–142. IEEE, 2018.
- [181] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018*, pages 129–140, New York, NY, USA, 2018. ACM. event-place: Amsterdam, Netherlands.
- [182] Jinxin Zhao, Jin Fang, Zhixian Ye, and Liangjun Zhang. Large scale autonomous driving scenarios clustering with self-supervised feature extraction. *arXiv preprint arXiv:2103.16101*, 2021.
- [183] Yue Zhao, Hong Zhu, Ruigang Liang, Qintao Shen, Shengzhi Zhang, and Kai Chen. Seeing isn’t believing: Practical adversarial attack against object detectors. *arXiv preprint arXiv:1812.10217*, 2018.
- [184] Ziyuan Zhong, Gail Kaiser, and Baishakhi Ray. Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles, 2021.
- [185] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. An empirical study on quality issues of production big data platform. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 17–26. IEEE Press, 2015.
- [186] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 347–358, 2020.
- [187] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium*, page 13, 2012.