Security Enhancement of Vehicle Software Systems

by

Lama J. Moukahal

A thesis submitted to the
School of Computing
in conformity with the requirements for
the Degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
December, 2021

Copyright © Lama J. Moukahal, 2021
Abstract

In an era of connectivity and automation, the vehicle industry is adopting various technologies to transfer driver-centric vehicles to intelligent mechanical devices driven by software components. However, software integration and network connectivity inherit numerous security issues. This thesis offers methods and tools that collaboratively enhance vehicle software security, making vehicles more resilient to cyber incidents. The uniqueness of Connected Autonomous Vehicles (CAVs) invites challenges for Vehicle Software Engineering (VSE) that render traditional software development models and practical solutions less effective for automotive software development. This research presents a Secure Vehicle Software Engineering (SVSE) lifecycle that ensures security-by-design, devoting security considerations throughout all phases of the vehicle software development process. We also introduce novel security enhancement techniques to be employed during the SVSE lifecycle. We propose security vulnerability metrics tailored to identify complexity within vehicle software systems that open the door for malicious behavior. These metrics are utilized with grey-box fuzzing to offer a vulnerability-oriented fuzz testing (VulFuzz) framework explicitly designed to address vehicle security testing challenges. Using the vulnerability scores, VulFuzz systematically directs and prioritizes the fuzz testing toward the most vulnerable components. Depending on the component under test, fuzz testing may
not be sufficient to assure a reliable system. Fuzz testing blindness prevents it from exploring the deep paths of the system, which is critical to evaluate for safety-critical components. As a result, we present a hybrid fuzz testing framework (VulFuzz++) that unites the efficiency of fuzzing and the precision of concolic execution to provide the automotive industry a reliable security testing tool. VulFuzz++ utilizes a tailored, targeted concolic engine that limits the symbolic exploration to only specific functions. While security testing can identify many vulnerabilities and enhance security, vehicles’ resilience against attacks might change during their operational lifespan. We introduce a security decay assessment framework that monitors vehicles’ security risks and recognizes security failure. We have implemented and evaluated the security enhancement techniques on OpenPilot, an automotive Autopilot system. The results show the effectiveness of the proposed techniques in strengthening vehicles’ resilience by identifying vulnerabilities at an early stage.
Acknowledgments

Foremost, I owe a deep sense of gratitude to my supervisor, Prof. Mohammad Zulkernine, for his invaluable guidance, intellectual stimulus, and sincerity. From inception to completion, thank you for keeping me motivated, acknowledging my challenges, and supporting me. Thank you for caring about the comfort of every member of our lab and for providing the utmost support you can. The knowledge and skills I learned from you are not bound to completing this thesis, and I will carry them throughout my professional carrier.

I also have to extend my gratitude to Martin Soukup, who I continued to work with after my internship at Irdeto. Martin’s constructive feedback and solid vision definitely enriched my research. For my supervising committee, Prof. Hossam Hassanein and Prof. Jana Dunfield, I am incredibly grateful for your assistance and suggestions throughout my Ph.D. I am also sincerely thankful to my examiners, Profs. Katerina Goseva-Popstojanova, Jianbing Ni, Juergen Dingel, and Sidney Givigi for dedicating time out of their busy schedules for my thesis examination and their constructive feedback. I want to extend my appreciation to Profs. Hassanein, Zulkernine, and all the faculty and staff members at Queen’s school of computing for actively enhancing the graduate program.

I am lucky to have a wonderful family who supports me in every step I take,
and I owe a deep sense of gratitude to them for helping me during my Ph.D. I am thankful to my parents for raising the person I am today. I have lost my father, Jamal, a long time before starting this Ph.D., but the principles I have learned from you empowered me in every step throughout this journey. I cannot thank my mother, Mounia, enough for always supporting my siblings and me in every decision we make. For encouraging us to dream high constantly. Words cannot describe enough the deep gratitude and appreciation I owe to my husband, Georges. Simply, I cannot imagine myself achieving this thesis without your help and encouragement. Thank you for the many times you listened to me, acknowledged my problems, prepared the best environments, and provided the best assistance. You are the ideal supporter, and I am grateful for having you by my side.

I am also thankful for having incredible siblings who never failed to help me. My brother, Ziad, always gives the proper support and has the right words that boost my confidence. My sister and best friend, Layal, provides me with the best tenderness, care, and love that any person needs to keep going in life. One of the challenges that started with my Ph.D. was being away from you, perhaps when we needed each other the most, yet you always managed to lift me. I would also like to express my appreciation to the youngest member of our family, my nephew, Abed Allah, who is too young to understand how supportive he was during my Ph.D. Yet your cute videos made me smile when nothing else would have cheered me up.

I would like finally to extend my appreciation to all our lab members for having the best workspace and support group. I specifically would like to thank Akib Anwar, Anika Anwar, and Farnood Faghihi. We started our graduate studies together, and I am thankful for making me feel surrounded by caring people.
Statement Of Originality

I hereby certify that all of the work described within this thesis is the author’s original work. Any ideas and techniques from the work of others are fully acknowledged following the standard referencing practices. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I authorize Queen’s University to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for scholarly research. I understand that my thesis may be made electronically available to the public.

Lama Moukahal

December, 2021
Co-Authorship


# Contents

Abstract i  

Acknowledgments iii  

Statement Of Originality v  

Co-Authorship vi  

Table of Contents viii  

List of Tables xii  

List of Figures xiv  

Glossary of Abbreviations xvi  

Chapter 1: Introduction 1  
  1.1 Motivations and Objectives 2  
  1.2 Contributions 6  
  1.3 Thesis Outline 9  

Chapter 2: Background and Literature Review 11  
  2.1 Vehicle Software Development 12
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Mapping of challenges to solutions</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Advantages of the V-Model and the Agile model</td>
<td>54</td>
</tr>
<tr>
<td>3.2</td>
<td>Disadvantages of the V-Model and the Agile model</td>
<td>55</td>
</tr>
<tr>
<td>3.3</td>
<td>Mapping of challenges to phases</td>
<td>55</td>
</tr>
<tr>
<td>3.4</td>
<td>Mapping of challenges to VSE models</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>OpenPilot inputs and outputs</td>
<td>85</td>
</tr>
<tr>
<td>4.2</td>
<td>OpenPilot bugs by years</td>
<td>86</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison between vehicle metrics and other metrics</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparative overview of VulFuzz, AFL, and the unguided mutation fuzzer running results</td>
<td>105</td>
</tr>
<tr>
<td>5.2</td>
<td>Crash analysis on different OpenPilot subprocesses</td>
<td>115</td>
</tr>
<tr>
<td>5.3</td>
<td>Average results of 10 runs</td>
<td>115</td>
</tr>
<tr>
<td>5.4</td>
<td>Results of the ablation study on VulFuzz engines</td>
<td>116</td>
</tr>
<tr>
<td>6.1</td>
<td>Prioritization and branch coverage during stage two and three</td>
<td>139</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparison between the results of VulFuzz++ and other methods</td>
<td>144</td>
</tr>
<tr>
<td>7.1</td>
<td>Attacker threat parameters</td>
<td>157</td>
</tr>
<tr>
<td>7.2</td>
<td>Impact level parameters</td>
<td>159</td>
</tr>
</tbody>
</table>
7.3 Vulnerability analysis of OpenPilot . . . . . . . . . . . . . . . . . . . . . 161
List of Figures

3.1 Vehicle engineering lifecycle ........................................ 48
3.2 Overview of the Secure Vehicle Software Engineering (SVSE) lifecycle 59
3.3 Security activities of the SVSE lifecycle mapped to the challenges and
the security standards .................................................... 62

4.1 Relationship between average metrics ratio and number of vulnerabilities 89

5.1 Vulnerability-oriented fuzz testing framework ........................ 93
5.2 An example call graph showing a path to a vulnerable function . . . 101
5.3 Comparison between the vulnerability-oriented mutation engine and
AFL’s mutation engine on a sample input ................................ 107
5.4 Comparison of statement coverage with respect to time in seconds . . 108
5.5 Comparison of crashes with respect to time in seconds .................. 110
5.6 The number of unique crashes and the number of times test cases
execute the weighted functions ........................................... 113
5.7 Venn diagram showing a comparison between unique crashes detected
by the different fuzz testing tools ........................................ 114

6.1 The VulFuzz++ framework composed of three stages .................. 127
6.2 Coverage analysis of VulFuzz++ by stages with respect to time in seconds 140
6.3 Crash analysis of VulFuzz++ by stages with respect to time in seconds 142
6.4 The number of unique crashes and the number of times test cases execute weighted functions 146
6.5 The number of unique crashes and running time 147
7.1 Autonomous Vehicle Security Decay Assessment (AVSDA) framework 152
Glossary of Abbreviations

ACC  Adaptive Cruise Control.

AFL  American Fuzzy Lop.

ALC  Automated Lane Centering.

ASPICE  Automotive Software Performance Improvement and Capability Determination.

AT  Attacker Threat.

AUTO-ISAC  Automotive Information Sharing and Analysis Center.

AUTOSAR  AUTomotive Open System ARchitecture.

AVSDA  Autonomous Vehicle Security Decay Assessment.

CAL  Crypto Abstraction Library.

CAVs  Connected Autonomous Vehicles.

CC  Component Coupling.

CG  Call Graph.
CM Component Maturity.

CPSs Cyber-Physical Systems.

CSM Crypto Service Manager.

CX Code Complexity.

DART Directed Automated Random Testing.

DM Driver Monitoring.

DR Data Risk.

DV Data Vulnerability.

ECUs Electronic Control Units.

EVITA E-safety Vehicle Intrusion Protected Applications.

FCW Forward Collision Warning.

FIFO First Come First Served.

FMEA Failure Modes and Effects Analysis.

FMEDA Failure Modes Effects and Diagnostic Analysis.

FN False-negative.

FP False-positive.

GPS Global Positioning System.
HIL  Hardware in the Loop.


IL  Impact Level.

IoT  Internet of Things.

ISO  International Organization for Standardization.

LDW  Lane Departure Warning.

LOC  Lines of Code.

LTS  Long Time Support.

MSDL  Microsoft Security Development Lifecycle.

ND  Nesting Depth.

NIDS  Network Intrusion Detection System.

NOC  Number of Children.

OEMs  Original Equipment Manufacturers.

OSs  Operating Systems.

PSI  Past Security Issues.

PWM  Pulse Width Modulation.

QA  Quality Accelerated.
RTM  Requirement Traceability Matrix.

SDLC  Software Development lifecycle.

SecOC  Secure Onboard Communication.

SLOC  Source Lines of Code.

SMT  Satisfiability Modulo Theories.

SPMT  Start, Predict, Mitigate, and Test.

SR  Security Risk.

SSDF  Secure Software Development Framework.

SSDL  Secure Software Development Lifecycle.

SVSE  Secure Vehicle Software Engineering.

SWC  Software Component.

SWCs  Software Components.

TARA  Threat Analysis and Risk Assessment.

TN  True-negative.

TP  True-positives.

TPT  Time Partition Testing.

UDS  Unified Diagnostics Services.
UNECE United Nations Economic Commission for Europe.

VSE Vehicle Software Engineering.

Chapter 1

Introduction

Connected Autonomous Vehicles (CAVs) are the future of transportation. It is projected that by 2024 the number of autonomous or semi-autonomous vehicles on the roads will surpass 54 million [1]. CAVs promise a future of safe transportation by eliminating the primary cause of vehicle accidents, drivers’ negligence.

Rapid progression in technology and network connectivity have changed the shape of vehicles. Modern automobiles are not just mechanical devices controlled and driven solely by humans. They are Connected Autonomous Vehicles (CAVs) that combine infrastructure and computer processing with advanced wireless communication to make decisions and provide drivers and passengers with a safer and more entertaining experience. Autonomous vehicles support intelligent features such as advanced driver assistance, real-time traffic alerts and responses, fast data transfers, and enhanced vehicular communication. These features add intelligence to mechanical vehicles at the cost of typically implementing a hundred million lines of software code [2].

Software integration and connectivity enable vehicles to be intelligent devices. However, this opens the window for software defects and vulnerabilities that attract malicious behavior. In fact, vehicles with both human drivers and autonomous driving
or driver assistance features pose the greatest risk due to the maximized attack surface compared to fully manual, disconnected vehicles or fully autonomous vehicles. Internet exposure introduces a plethora of vulnerabilities and facilitates attackers’ jobs. Hackers’ threats in the vehicle’s domain are not limited to a breach that only exploits personal data; they can amplify the risk by altering the vehicle software systems. There are currently many reported vehicle attacks initiated against different vehicle manufacturers [3].

Accordingly, Original Equipment Manufacturers (OEMs) are striving to enhance their security measures to increase vehicles’ resilience to cyberattacks. However, ensuring resilient vehicles is an unattainable goal without incorporating a comprehensive approach to cybersecurity. Mandated by the Road Vehicles Cybersecurity Engineering standard ISO/SAE 21434 [4], automotive systems should be designed to be foundationally secure by embedding cybersecurity considerations and solutions into all the vehicle Software Development Lifecycle (SDLC) phases.

In an attempt to aid the automotive industry in enhancing vehicles’ resilience, this thesis focuses on the security of automotive software systems. Automotive systems have different architecture and requirements from other software systems [5–7]. *Hence, a comprehensive understanding of automotive systems’ unique characteristics and software development process is needed to design the appropriate solutions that assure security and manage the challenges of automotive systems.*

1.1 Motivations and Objectives

This thesis strives to provide practical and comprehensive solutions that manage the unique architecture of CAVs and strengthen the security of automotive software
systems, making CAVs more resilient to cyberattacks.

We identify the unique research challenges originating in Vehicle Software Engineering (VSE). We also compare and contrast how existing standards, projects, tools, languages, and research approaches address these challenges. Our analysis shows that automotive systems require a comprehensive approach to cybersecurity.

Since modern vehicle development depends on software, securing the development life cycle is vital to provide consumers with better experiences. Different standards like AUTomotive Open System ARchitecture (AUTOSAR) [8], J3061 [9], and the International Organization for Standardization (ISO) [10] 26262 [11] highlight the importance of deploying security measures during all the phases of vehicle software engineering (VSE) [12]. As the need for developing secure vehicle software systems is higher than ever, ISO collaborated with the Society of Automotive Engineers (SAE) [13] and designed a standard, ISO/SAE 21434 [4], that targets explicitly secure development.

These standards are essential to advise OEMs during VSE and formulate a security norm in the vehicle industry. Nevertheless, following the regulations of all these standards while managing the unique architecture of CAVs, make the Secure Software Development Lifecycle (SSDL) challenging for automotive systems. The automotive industry requires a comprehensive SSDL that incorporates special considerations to manage all the challenges of automotive cybersecurity development while ensuring compliance with international security standards. To guide the security engineers, this thesis presents a Secure Vehicle Software Engineering (SVSE) lifecycle that incorporates security activities that guarantee compliance with international security standards and assure software security.
1.1. MOTIVATIONS AND OBJECTIVES

More importantly, this industry still requires practical solutions that can be deployed during all the phases of SSDL to verify the system’s security and avoid catastrophic incidents. The lack of quality assurance and testing procedures in the vehicle industry is one of the primary factors contributing to the existence of vulnerabilities [14]. Clearly, security testing is a crucial phase in any SSDL to identify vulnerabilities and system weaknesses. Different security assurance methods are utilized in the vehicle industry, including static code analysis, dynamic program analysis, vulnerability scanning, penetration testing, and fuzz testing [15]. These security testing techniques diminish the vulnerabilities in a system.

Regardless, security testing for vehicle software systems is a complex task that leaves OEMs with multiple challenges [12]. The vehicle software system is a complex system with around a hundred million lines of code residing and running on dozens of Electronic Control Units (ECUs) [16]. These ECUs operate based on inputs from radars, lidars, cameras, ultrasonic sensors, temperature sensors, tire pressure sensors, and many other sensors. As vehicles operate in a continuously evolving environment, inputs to ECUs vary drastically. Hence, it is difficult to predict all possible input combinations of ECUs. The automotive industry requires robust security evaluation solutions that manage the vehicle security testing challenges while identifying possible loopholes and weaknesses that leave the automotive system vulnerable to malicious cyberattacks.

Our research addresses these requirements by enhancing security assurance in the vehicle industry. This thesis reviews the difficulties of security assurance during VSE and proposes several security enhancement techniques that handle the challenges and
meet security validation requirements. One major challenge that faces security engineers during the security testing process is the size and complexity of automotive systems. With a limited time budget, assuring a comprehensive evaluation of the system components is an infeasible job. Thus, ensuring an effective and reliable security assurance process requires prioritizing testing and giving more attention to weak components that expose the vehicle to greater risk. This thesis aids security engineers by providing security vulnerability metrics that consider the unique architecture of autonomous system while identifying the components that pose a high risk on the system.

Security specialists recommend fuzz testing to identify vulnerabilities within vehicle software systems. This thesis presents a vulnerability-oriented fuzz testing framework (VulFuzz) that utilizes the designed security vulnerability metrics to direct and prioritize the fuzz testing toward the most vulnerable components. While most grey-box fuzzing techniques aim solely to expand code coverage, the proposed approach assigns weights to ensure a thorough examination of the most vulnerable components.

Grey-box fuzz testing can validate the system with various scenarios, but it might fail to explore the deep paths of automotive systems. For safety-critical components, a comprehensive validation is required to prevent dramatic incidents. Hence, the automotive industry needs a reliable security testing tool that dynamically explores the system and assures a comprehensive evaluation. This thesis also presents a hybrid fuzz testing framework (VulFuzz++) that leverages grey-box and white-box testing capabilities while lessening the shortcomings of each testing technique, granting the automotive industry a trustworthy security testing tool.

Security within automotive systems cannot be restricted to the development phase
only; vehicles operate on the roads for over ten years [17]. Over this long lifespan, new software vulnerabilities arise and adopted security practices become incompetent. Thus, it is crucial to monitor the changing threat level of vehicles during their entire lifespan. Traditional threat and risk assessment methods [18] determine security risks by identifying and classifying all possible attack scenarios, which is labor-intensive. This thesis offers a security decay assessment framework that quantitatively evaluates the automotive system without any additional overhead.

The security enhancement techniques presented in this thesis can recognize vulnerability and weaknesses in the automotive software system before they expose themselves in the form of attacks, making CAVs safer and more reliable.

1.2 Contributions

The work presented in this thesis is needed in the automotive industry to facilitate security engineers’ jobs, comply with international security standards, identify security vulnerabilities, and recognize security decay. The main contributions of this thesis are summarized as follows.

1. Secure Vehicle Software Engineering (SVSE) Lifecycle. This thesis defines Vehicle Software Engineering (VSE) and provides an in-depth and comprehensive analysis to perceive existing software engineering processes detailing their strengths and limitations in the context of CAVs. Moreover, we present a comprehensive Secure Vehicle Software Engineering (SVSE) lifecycle that capitalizes on experiences from traditional security development lifecycles and ensures the security-by-design of automotive systems. The SVSE lifecycle incorporates security activities that
mitigate the development and operation challenges, reducing cybersecurity violations. It assists the automotive industry in complying with international security standards by granting security considerations throughout the development lifecycle that accommodate the requirements of industrial standards. The SVSE lifecycle promises manageability and deliverability of security practices throughout the full-life span of vehicles, making CAVs more reliable [12,19].

2. **Vehicle Security Vulnerability Metrics.** We guide security engineers by providing security vulnerability metrics that identify automotive systems’ weak or vulnerable components while considering CAVs architecture. The early indicators of vulnerabilities are (1) Code Complexity, (2) Component Coupling, (3) Input and Output Data Vulnerability, (4) Past Security Issues, and (5) Component Maturity. These security metrics quantitatively measure the vulnerability of each component in the automotive software system. We construct a calculation methodology and a weighted equation for each metric. Automatic assessment of weak components assists security engineers in prioritizing the components based on their security vulnerability [20,21].

3. **Vulnerability-Oriented Fuzz Testing Framework.** We present a vulnerability-oriented fuzz testing framework (VulFuzz) that manages many of VSE’s testing challenges efficiently and reliably. The proposed grey-box fuzz testing framework defeats common blind black-box fuzzers in the vehicle industry by knowledgeably validating the system. VulFuzz systematically prioritizes the testing toward weaker components of the vehicle software systems. The framework utilizes the designed security vulnerability metrics to identify vulnerable components in the vehicle software systems and ensures thorough testing of these components by
assigning weights. Hence, we strengthen vehicles’ resilience against unpredicted cyberattacks. Moreover, to account for the strong input validation of autonomous systems, we introduce a mutation engine that performs small data type mutations at the inputs’ high-level design. Hence, VulFuzz improves the grey-box fuzzing performance by reducing dropped fuzz messages [22].

4. *Hybrid Fuzzing Framework for Vehicles.* We provide a hybrid fuzz testing framework (VulFuzz++) that employs fuzzing and concolic execution concurrently, taking advantage of the agility of fuzzing and the accuracy of concolic exploration. VulFuzz++ evaluates the system’s deep and vulnerable paths without increasing testing complexity. We manage the limitations of concolic execution by designing a hybrid fuzzing framework that offloads the work to grey-box fuzzers and limits the scope of symbolic exploration. We introduce a prioritized and targeted concolic exploration approach that systematically satisfies input-specific conditions without symbolically exploring the entire execution path. We augment VulFuzz vulnerability identification and outperform traditional fuzz testing techniques, illustrating the effectiveness and robustness of VulFuzz++ [23].

5. *Autonomous Vehicle Security Decay Assessment Framework.* We present an Autonomous Vehicle Security Decay Assessment (AVSDA) framework that analyzes and predicts the system’s security risk over vehicles’ lifespan. The framework analyzes vulnerable software components periodically and estimates the security risk level to identify security decay. AVSDA employs the proposed security metrics to automatically identify potentially weak components and quantify security risk [21].
1.3 Thesis Outline

The remainder of this thesis is organized as follows.

- **Chapter 2, Background and Literature Review:** This chapter provides knowledge about existing standards, tools, projects, and research efforts that attempt to mitigate the unique challenges of Vehicle Software Engineering (VSE). It also identifies the security assurance challenges and reviews the current existing security assurance techniques in the automotive industry.

- **Chapter 3, Engineering of Vehicle Software Systems:** This chapter is divided into two main parts. The first examines existing software engineering processes in the automotive industry and presents a thorough analysis of the advantages and disadvantages of each process. The second part of this chapter introduces the Secure Vehicle Software Engineering (SVSE) lifecycle. It discusses the importance of each phase of the SVSE and details the security activities of the SVSE lifecycle.

- **Chapter 4, Security Vulnerability Metrics for CAVs:** This chapter introduces the designed security vulnerability metrics. It discusses the usefulness of each metric in identifying the system weaknesses and describes the metrics’ calculation method.

- **Chapter 5, Grey-box Fuzz Testing for Vehicle Systems:** This chapter introduces the Vulnerability-oriented fuzz testing framework (VulFuzz). First, it describes the framework design and introduces the engines of the framework. Then, this chapter outlines the experimental evaluation and presents the results.
• **Chapter 6, Hybrid Fuzz Testing for Vehicle Systems:** This chapter introduces the hybrid fuzz testing framework. It describes the motivation behind using concolic execution and explains the details of the targeted concolic engine. The chapter ends by explaining the experiments and the results.

• **Chapter 7, Security Decay Monitoring:** The chapter presents the Autonomous Vehicle Security Decay Assessment (AVSDA) framework. It explains the phases of the framework and illustrates its usefulness in identifying the security decay of automotive systems.

• **Chapter 8, Conclusion:** This chapter summarizes the thesis contributions, presents the limitation of our work, and introduces some future research directions.
Chapter 2

Background and Literature Review

This chapter starts by providing background knowledge about vehicle software development. It identifies the unique characteristics of vehicle software systems and discusses the challenges of Vehicle Software Engineering (VSE). It surveys existing standards, tools, projects, and research efforts and maps the identified challenges to existing solutions while discussing their capabilities in mitigating the challenges of VSE. We notice a need for further security assurance solutions to be adopted during VSE. This chapter then provides an overview of vehicle security assurance. It identifies the security assurance challenges within vehicle software systems and thoroughly reviews existing security assurance techniques highlighting the strengths and shortcomings of each. Finally, we review related fuzz testing techniques not adopted in the automotive industry but offer a robust security validation technique for other software applications.
2.1 Vehicle Software Development

2.1.1 Uniqueness of Vehicle Software Systems

Vehicle software systems differ from other software systems. We identify the unique characteristics that make vehicle software systems different from their conventional counterparts in other domains as described in the following paragraphs.

Wide Range of Users. Vehicle software systems are known for their longevity which adds a necessity to support a variety of users [7]. During their lifetime, broad ranges of end-users interact with these systems. Consumers with a variety of backgrounds, knowledge, and expectations interact with the system to run, update, customize, and maintain its features. For example, vehicle technicians use the system to check updates and run diagnostic reports that can assist them with their work. In contrast, drivers interact with the system to customize the running features and make their driving experience more enjoyable.

Large Number of Hardware Components. Connected Autonomous Vehicles (CAVs) contain more than 100 Electronic Control Units (ECUs); all of these ECUs are bridged together and exchange information to provide some services or features. This large number of communicating ECUs require a tailored middleware and increase the software complexity [5].

Sensitive System. CAVs are deployed in a critical environment. The vehicle system is not only responsible for running and providing a certain functionality but is also expected to ensure safety for drivers, passengers, other vehicles, and pedestrians. Thus, any glitch in the system can have a life-threatening impact. This puts higher pressure on software engineers who are expected to develop a trustworthy systems [7].
Heterogeneous Functions. Transferring a vehicle’s controls to a smart mobile device requires adding various intelligent Software Components (SWCs) that operate thousands of mechanical devices. These SWCs are very diverse, spanning five categories: 1) multimedia and telematics, 2) body software, 3) safety software, 4) power train and chassis control software, and 5) infrastructure software [24]. Different components in a vehicle software system communicate and share information to provide better features for drivers. For example, the safety system communicates with the central locking system in a vehicle to unlock doors in case of an emergency to ease the rescue procedure of passengers. For this reason, managing communications between heterogeneous functions is a distinctive feature to consider when developing vehicle software systems.

Divergence of Maintenance. A vehicle can operate for more than three decades, which implies that vehicle software systems are subject to maintenance at least once [24]. The maintenance process of vehicle software systems is a critical task. In most cases, the system is not repaired by the engineers who built it. Vehicle owners tend to fix their vehicles at the nearest garage or service center. Thus, vehicle engineers lose control to maintain the software system as various technicians with different skills and knowledge are involved in accomplishing the maintenance task. Another fact about vehicle maintenance is the nature of ECUs. Vehicle software systems run on ECUs which have a short lifespan. It is estimated that 25% of ECUs in a vehicle will have to be replaced after 3 years of their production [24]. Note that ECU development is advancing and changing quickly, replacing an ECU with the same one is less common. This adds more unique characteristics to the maintenance process as the engineers responsible for this task have to not only replace non-functioning ECUs but also verify
that functionalities installed on these ECUs are compatible with new hardware.

**Different Communication Means.** Unlike most software systems, vehicle software systems are exposed to different communication means. Although this brings many advantages, it also increases complexity. Vehicle engineers are expected to handle various networking protocols and manage smoothly the adaptability of the system to different domains [5].

### 2.1.2 VSE Challenges

To maintain the performance, security, and reliability of vehicle software systems, software engineers are expected to practice a software engineering model that best fits the unique characteristics of vehicle software systems. Since vehicle software systems have different characteristics than other software systems, building them requires certain expertise and high skills. With the growing importance of software in vehicles and the high demand for innovative features, vehicle engineers are left with many complex challenges that require attention from different specialists. This section reviews the following challenges originating in Vehicle Software Engineering (VSE).

**Software Integration.** A vehicle is no longer an assembled mechanical part only, it is also an integrated system. Vehicle systems have many different functionalities that require different kinds of expertise. For this reason, a large part of software development is outsourced in the vehicle industry. Although this might have many advantages, it also introduces several challenges and increases the complexity of VSE. Various components developed by different entities must be integrated into one system that is capable of handling information exchange between functionalities easily.
and quickly [7]. Thus, software integration should be well planned and managed at an early stage of the development lifecycle to ease the integration process. A software engineer should consider hardware specifications, programming languages, and communication protocols to ensure compatibility between an outsourced functionality and other components of the system. Other factors that contribute to the complexity of vehicle software integration are the heterogeneous functionalities and diverse hardware components that are involved in the systems. The wide range of technologies incorporated in a vehicle system raises the challenges of software integration and makes software engineers’ jobs more complex [24].

**Communication Diversity.** Today’s modern vehicles can communicate with external environments. For example, a connected vehicle can communicate with a transportation system to gather information about road congestion. Connected Autonomous Vehicles (CAVs) incorporate different communication technologies, including Vehicle to Vehicle (V2V), Vehicle to Infrastructure (V2I), User to Vehicle (U2V), and Intra-Vehicle (intra-V) communication. Each of these communication means uses different networking protocols [25]. This diversity increases the difficulties of VSE since software engineers are expected to manage the diversity and handle it during all phases of the development lifecycle to ensure proper and efficient communication between different Electronic Control Units (ECUs), sensors, vehicles and systems.

**Compatibility and Code Reusability.** A vehicle lifespan is considerably longer than an ECU lifespan. Thus, within the lifecycle of a vehicle, ECUs (hardware) are expected to be updated or even replaced. Since code is usually designed and developed for a specific processor, replacing an ECU brings another challenge in VSE. The software component residing on an ECU has to be compatible with the hardware
2.1. VEHICLE SOFTWARE DEVELOPMENT

that is being repaired or will be replaced. This is only achievable if the compatibility is addressed during VSE phases [5]. In addition to compatibility, a code is expected to be reusable between different vehicle designs to diminish development costs. A major portion of functionalities in a vehicle software system are either slightly changed or reused. Hence, developed features have to support code reusability. This task requires efficient planning during the VSE to handle code reusability while maintaining a good level of code optimization [24].

Testing Complexity. Software testing is a key process in any software engineering model. In fact, in a safety-critical system, testing is an essential step to maintain the integrity of the product and to ensure safety. According to our findings, the vehicle industry is witnessing three main complexities during the software testing phase: (1) Test-bed, (2) Cross-ECU Testing, and (3) Input and Output Validation. We explain further the complexity of testing in Section 2.2.

Maintenance and Error Recovery. In the vehicle industry, long-term maintenance is considered one of the most challenging requirements [5, 24]. The high number of functionalities and different technologies in a vehicle make a vehicle software system a complex one. More importantly, this complex system is also safety-critical. Hence, any update or fix in the system has to be accomplished smoothly and without leaving any possible hazard. An update in one ECU has to be compatible with the entire system as ECUs are interconnected and depend on each other. Moreover, in case of any failure, prompt recovery should take place. However, finding the root of the problem and fixing it is a challenging task for vehicle software systems. At the level of a Central Processing Unit (CPU), error logging takes place. In any failure situation, many systems incorporate hardware redundancy to ensure reliable and
continuous service. Nevertheless, there is no systematic error recovery beyond CPUs and hardware redundancy is still limited for vehicle systems. For this reason, error recovery is considered a challenging task that requires a well-regulated solution.

**Safety and Reliability Assurance.** Without any doubt, safety is the most important criteria in any vehicle system. Most of the functionalities included in a vehicle system aim to increase the safety of passengers. For example, the *Crash Detection* system monitors the surroundings of a car to alert drivers of any potential collision. In advanced systems, the *Crash Detection* can even take action and stop the car to prevent accidents. Though theoretically modern vehicles are safer, assuring this safety is a complex job. This is due to the smart features that step into vehicle systems and make it a complex one [26]. System engineers have to guarantee that the system cannot be in a state that leads to hazards. Handling the safety and reliability of vehicle software systems during the development lifecycle is a daunting job that can always be questionable without following a solid standard that facilitates safety assurance.

**Software Security and Data Privacy.** Increasing software integration in modern vehicles amplifies security vulnerabilities in vehicles and leads to more security issues. Many reported security attacks are successfully initiated on vehicle software systems [27]. As a security threat can endanger the lives of passengers, software security becomes a critical element in VSE. Recognizing attacks and vulnerabilities during the initial phases of the development lifecycle is a key to a more secure system. Considering the wide list of attack vectors, offering authenticity, integrity, and confidentiality for vehicle software systems becomes a complex step [28]. Vehicle software systems are different from other software systems and existing security solutions can
be limited when deployed in the vehicle industry [29].

Moreover, to mimic human sense and intelligence, modern vehicles collect and analyze data to provide smarter functionalities. Although this can enhance passengers’ experience, it leaves the drivers’ and passengers’ privacy under question [30]. Vehicle systems collect different types of data; for example, internal vehicle information is collected from radars, lidars, cameras, ultrasonic sensors, temperature sensors, tire-pressure sensors, and many other sensors. While such kind of information does not jeopardize drivers’ privacy, vehicle systems collect information that can reveal some personal and private information about drivers. Some of the private data include drivers’ navigation preferences, locations traversed, stores visited, drivers’ contact information, and biometrical data [31]. This kind of information should be protected during the VSE process. However, considering the complexity of the system and the enormous amount of data collected, this job becomes challenging to manage while maintaining system optimization.

### 2.1.3 Comprehensive Analysis of Existing VSE Solutions

In what follows, we analyze the existing practical solutions in the light of each research challenge identified in Subsection 2.1.2. Table 2.1 depicts the mapping according to the services provided by each solution and the challenges they can solve. As outlined in the table, we group current solutions into four categories according to the nature of the work: a) standards and processes, b) tools and projects, c) model-based languages, and d) other research work. The first category Standards and Processes, groups international policies that aim to organize the work of Vehicle Software Engineering (VSE). The second category, Tools and Projects, covers relevant
tools that can be utilized within VSE to lessen difficulties. Model-Based Languages is a category that holds the languages that were created specifically for vehicle software systems. Finally, the last category, Other Research Work, includes the most relevant studies that target the challenges of VSE.

**Software Integration.** Bringing together different components and functionalities into one vehicle software system is one of the most challenging tasks of VSE. Ensuring that various subsystems, which are commonly outsourced to different companies, can successfully integrate and exchange information requires careful planning.

The most powerful approach for this challenge is adopting a standard that defines and unifies the software integration process and present its alignment within the development lifecycle. One of the most commonly known standards in the vehicle industry is the AUToMotive Open System ARchitecture (AUTOSAR) standard [8] that promises to simplify system and software integration. The standard achieves this by ensuring that each phase of VSE is prepared with the needed policies to reach successful integration. For example, it provides naming rules for each entity in the system to be followed during the requirement phase. This can help in successfully ensuring proper communication between different elements of the system. Moreover, the standard layered architecture dedicates a run-time environment layer to ensure smooth communication between heterogeneous functionalities. AUTOSAR basic software layer allows the separation of Software Components (SWCs) related to functionalities from ECU infrastructure SWCs. Thus, the same functionalities can be integrated easily into different ECU hardware [32,33]. Other international practices that support system integration for vehicle software systems are the Motor Industry Software Reliability Association (MISRA) [34] and Automotive-Software Process Improvement
and Capability Determination (Automotive-SPICE) [35]. Even though MISRA does not target system integration directly, offering code practices and consolidating the work among different parties can facilitate the process of system integration.

In practice, some tools support AUTOSAR structure and MISRA practices, which can help in making software integration more manageable. Some of these tools are Embedded Coder by MathWorks [36] and Digital Signal Processing and Control Engineering (dSpace) [37] which aim to support automatic code generation following AUTOSAR Standard. CHESS [38] offers a requirement validation tool that can check the defined specification against incompatibility.

As explained earlier, software integration has to be planned during the initial phases of the development lifecycle. For this to be completed, languages like Electronic Architecture and Software Tools Architecture Description Language (EAST-ADL2) [39] can facilitate the requirement specification phase. EAST-ADL2 is designed particularly for vehicle software systems to enable system modeling, decrease the complexity, and introduce abstraction to the system. EAST-ADL2 is successfully capable of unifying the work between different parties that are involved in development. In addition, EAST-ADL2 supports the AUTOSAR standard, which makes it a good and useful solution for software integration [40]. Another model-based language that can be helpful in this challenge is the Automotive Modeling Language (AML) [41].

Apart from standards and tools, some researchers show interest in targeting this challenge. For example, Holtman et al. [42] introduce a component-based hierarchical software platform to support vehicle software systems. Li et al. [43] also follow AUTOSAR architecture framework and use model transformation techniques. The
2.1. VEHICLE SOFTWARE DEVELOPMENT

Table 2.1: Mapping of challenges to solutions (\(\square\) shows full effectiveness, \(\▲\) shows partial effectiveness, and \(\△\) shows limited effectiveness)

<table>
<thead>
<tr>
<th>Challenges</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software Integration</td>
</tr>
<tr>
<td>AUTOSAR [8]</td>
<td>(\square)</td>
</tr>
<tr>
<td>ISO 26262 [10]</td>
<td>(\▲)</td>
</tr>
<tr>
<td>MISRA [34]</td>
<td>(\△)</td>
</tr>
<tr>
<td>Automotive-SPICE [9]</td>
<td>(\square)</td>
</tr>
<tr>
<td>Standards and Processes</td>
<td></td>
</tr>
<tr>
<td>HIP-OPS [4]</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Embedded Coder [11]</td>
<td>(\△)</td>
</tr>
<tr>
<td>J3061 [9]</td>
<td>(\▲)</td>
</tr>
<tr>
<td>AUTOSAR [8]</td>
<td>(\square)</td>
</tr>
<tr>
<td>ISO 26262 [10]</td>
<td>(\▲)</td>
</tr>
<tr>
<td>MISRA [34]</td>
<td>(\△)</td>
</tr>
<tr>
<td>Automotive-SPICE [9]</td>
<td>(\square)</td>
</tr>
<tr>
<td>Tools and Projects</td>
<td></td>
</tr>
<tr>
<td>CANoe [8]</td>
<td>(\△)</td>
</tr>
<tr>
<td>J3061 [9]</td>
<td>(\▲)</td>
</tr>
<tr>
<td>CHESS [25]</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Traceroute [34]</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Another</td>
<td>(\△)</td>
</tr>
<tr>
<td>Model Based Languages</td>
<td></td>
</tr>
<tr>
<td>AML [40]</td>
<td>(\▲)</td>
</tr>
<tr>
<td>FAST-ADL2 [41]</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Other</td>
<td></td>
</tr>
<tr>
<td>Automotive-SPICE [9]</td>
<td>(\square)</td>
</tr>
<tr>
<td>Research Work</td>
<td></td>
</tr>
<tr>
<td>Albinet et al. [44] (2009)</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Lindlar et al. [45] (2009)</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Engel et al. [53] (2009)</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Biehl et al. [54] (2010)</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Izerrouk et al. [55] (2010)</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Oka et al. [56] (2018)</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Oka et al. [56] (2018)</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Fowler et al. [63] (2018)</td>
<td>(\▲)</td>
</tr>
<tr>
<td>Standberg et al. [64] (2018)</td>
<td>(\▲)</td>
</tr>
</tbody>
</table>

approaches followed by both studies can promote software integration by implementing AUTOSAR separation layers. Albinet et al. [44] contribute to the design phase of VSE by providing a methodology to transfer requirements to models. This approach can facilitate software integration if it is deployed among all the functionalities.

**Communication Diversity.** Diversity within vehicle software systems is not limited to functionalities. Vehicle software systems include various communication means and protocols which amplify the complexity of VSE. AUTOSAR is one of the standards that seeks a solution for this challenge. For example, this standard introduces the Protocol Data Units (PDU) router module [8]. The primary services offered by a PDU router are communication interface modules and communication manager. Moreover, the standard sets requirements for the gateway ECU, which aims to distribute data.
among different entities in the vehicle network. According to the requirements of the gateway, different models (e.g., PDU gateway, Transport Protocol (TP) gateway, and single gateway) can be applied. Though, in principle, AUTOSAR supports communication diversity in VSE, the increased requirements must be taken into account. AUTOSAR assumes that PDUs are defined identically in terms of length and content on both the source and target network. MISRA standard also contributes to this challenge; however, their contribution is less comprehensive. While covering the software engineering aspect of vehicles, MISRA considers communications and multiplexing management. Nevertheless, the standard focus is targeted towards code practices rather than communication diversity.

Li et al. [43] introduce in their research work a communication service to support different networking protocols of vehicle software systems. The proposed communication service hides lower layer protocol details to manage communications easily. The concept implemented by the researchers is derived from the AUTOSAR standard; however, their implementation only supports three kinds of networking protocols which are Controller Area Network (CAN), Local Interconnect Network (LIN), and FlexRay. For this reason, we consider that Li et al. have focused less on communication diversity challenge compared to other existing work.

**Compatibility and Code Reusability.** Developing an efficient code is one of the key elements for a successful product. To make this job easier, code reusability can boost efficiency and save time. Nevertheless, with the increased complexity of vehicle software systems, this job has become too complicated. One of the first objectives of AUTOSAR is to avoid frequent development of similar functionalities [65]. To achieve code reusability, AUTOSAR separates hardware and SWCs by introducing
decoupled architecture layers. This makes the functionalities independent of ECU hardware. Hence, this can boost reusability and compatibility as a SWC can be adopted in different projects and different hardware. MISRA policies tackle this challenge by offering code practices for embedded systems. Having a consolidated structure increases code portability and reusability. In fact, in January 2019, MISRA and AUTOSAR agreed on releasing a unified coding guidelines to ease the work further in the vehicle industry [34].

Currently, many tools support automatic code generation [36–38]. CHESS is a model-based tool where code configuration is performed based on the model design, making hardware-specific components easy to be tailored. Embedded Coder has a built-in feature that supports both AUTOSAR and MISRA standards. Moreover, dSpace follows AUTOSAR standard and develops its architecture. As explained earlier, following such standards while performing code generation can facilitate code reusability. Not only these tools rely on AUTOSAR, but also researchers use code synthesis techniques while following the standard to guarantee code compatibility and reusability [42,43].

**Testing Complexity.** AUTOSAR introduces the acceptance-testing module, which is tailored to match its architecture and requirements. The main aim of this module to lower the cost of testing. However, the presented testing coverage is not comprehensive, which entails extra efforts in VSE process to validate and verify the system [8]. Similarly, MISRA issued best practices for validating and verifying vehicle software systems [34]. However, we noticed that the report is not being maintained as MISRA’s current focus is targeted toward best code practices.

The International Organization for Standardization (ISO) provides a safety-compliant
standard (ISO 26262) [10] to ensure safety within vehicle systems. One of the identified terms of ISO 26262 is verification. The goal of this defined term is to detect safety anomalies. However, ISO 26262 just defines certain terms without explicitly advising how to apply validation and how to contain its complexity. Automotive-SPICE includes different engineering models that discuss software security [35]. It includes best practices for software construction verification, software integration testing, software testing, and system testing. The process summarizes the best practices to facilitate the testing phase. J3061 [9] provides general instructions for security testing. The standard discusses the best security testing methodologies for vehicle software systems. In general, existing standards and models refrain from proposing tools for testing and they do not construct a testing cycle that is specifically tailored for VSE.

In the industry, there are several tools that aim to automate and facilitate testing challenges. Nevertheless, most of these tools are targeting specific requirements. VectorCAST [47], which follows the recommendation of ISO 26262, is a software testing tool that can be used across different stages of development lifecycle. VectorCAST provides automatic regression testing, requirement validation, and input and output validation. The tool can also handle the challenge of cross-ECU testing as it can be executed on a simulator or an actual embedded system. However, VectorCAST does not validate the network management of vehicle software systems. On the contrary, CANoe [48], which is designed by the same company, specifically target ECU testing and examine the entire ECU network. Other testing tools [37, 38, 49, 51] provide specific testing services. For example, dSpace can support cross-ECU testing with Virtual ECUs (V-ECUs), which can replicate the behavior of an ECU to enable the
testing of a developed ECU.

Some researchers focus on constructing a test-bed to allow testing at an early stage of the development lifecycle. Ardila et al. [59] construct a test-bed that can reproduce the behavior of a vehicle network and monitor the exchanged messages on the CAN bus. Zheng et al. [60] offer a test-bed specific for security testing. Drolia et al. [16] design a comprehensive test-bed that can handle cross-ECU testing and network monitoring. Such a test-bed allows early verification of the system. Apart from test-bed construction, other researchers [58, 62, 63] present methods to perform input and output validation of an ECU by sending fuzzy messages. Albinet et al. [44] contribute to the testing challenge by providing a methodology to validate the mapping of requirements to the design phase. Such studies can contribute to the validation of the system to avoid defects in the final product.

**Maintenance and Error Recovery.** Within the vehicle industry, research is targeted toward developing a functional and safe system. Thus, a small number of researchers handle the maintenance and error recovery challenges. AUTOSAR offers a complete error handling chain, including identification and recovery techniques. Moreover, the standard suggests error handling mechanisms that best suits autonomous vehicles [8]. As error handling can affect the vehicle’s safety, ISO 26262 releases a set of maintenance and error handling recommendations. For example, ISO 26262 recommends the implementation of static recovery mechanisms for higher assurance of safety for vehicle software systems [11].

ANSYS Medini Analyze [50] offers a model-based analysis method and a design tool. It presents a traceability mechanism that navigates throughout the system to identify errors that can affect the safety. Though, as a safety solution, the tool
offers a solid analysis, we consider their effectiveness towards error handling and maintenance challenge as limited since they do not provide a complete chain that can contain the risks, but can only identify them. IBM Rational Logiscope [46] includes a code validation tool based on the MISRA standard. The validation tool has a minor contribution to address this challenge by eliminating unnecessary and redundant code, which can ease maintenance in later stages of the development lifecycle.

Li et al. [43] offer inadequate contribution in this area by proposing the addition of a stack monitor to identify possible errors that can be faced due to stack overflow. The researchers plan to extend their error avoidance methodologies to increase reliability.

**Safety and Reliability Assurance.** A number of research and development initiatives are taken to provide passengers with a safe experience. The most popular standard that focuses on this challenge is ISO 26262 [10]. The standard defines safety confidence levels to be followed by different Original Equipment Manufacturers (OEMs) and suppliers. One of the key components of ISO 26262 is Automotive Safety Integrity Levels (ASIL), which intends to analyze the system concerning possible threats. ASIL estimates the safety risks of passengers based on three main factors: (1) probability of exposure, (2) controllability of a driver, and (3) severity of an incident [11]. In the context of safety, MISRA contributes to ISO 26262 by providing guidance for the requirements allocation processes which can be deployed to assure functional safety. Moreover, J3061 emphasizes that safety cannot be guaranteed without securing the system. The standard builds over ISO 26262 and extends its policies to cover security. AUTOSAR aims to reduce the risks and increase the reliability of the system [32]. The standard advocates for handling functional safety from the initial phases of the development lifecycle as it may affect design and development.
decisions. However, AUTOSAR is not a safety standard. Thus, applying AUTOSAR standard alone cannot ensure the safety of the system [8].

Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [45] is a safety and analysis tool that performs automatic synthesis of fault trees, failure modes, and effect analysis. Papadopoulos et al. [66] find the tool to be capable of ensuring safety for complex systems like vehicle software system as it follows ISO 26262 standard. HiP-HOPS is not the only tool to follow ISO 26262 standard, Embedded Coder, dSpace, and CHESS take into consideration the safety requirements in their automatic code generators. Tools like VectorCAST, CANoe, and Time Partition Testing (TPT) [49] also follow ISO 26262 while performing full analysis for the code to validate its safety. Such mechanisms can support software engineers in VSE by making the validation process automated. IBM Rational Logiscope offers automatic code validation that helps in adopting international standards like ISO 9216 [67]. However, the tool does not support ISO 26262, which is specific for autonomous systems. Among the discussed tool, ANSYS Medini Analyze presents the most comprehensive solution. The tool performs hazard and operability analysis, fault tree analysis, failure modes and effects analysis (FMEA), and diagnostic analysis (FMEDA), all grouped in one automated tool. ANSYS Medini Analyze can be used during all the phases of the development lifecycle to comply with ISO 26262 standard.

Most of the presented tools perform the validation late during the development lifecycle [45, 47]. Hence, fixing the system defects becomes a cost-effective and time-consuming job. Izerrouken et al. [55] make use of formal methods to check the safety of the defined requirements. Moreover, Macher et al. [56] present a model-driven
engineering framework that is compliant with ISO 26262 and considers safety starting from the requirement phase. Other researchers contribute to the safety of the system during the design and implementation phases. Biehl et al. [54] introduce a model-based development toolchain that seamlessly integrates safety. Lindlar et al. [52] ensure code safety by using finite state machines.

**Software Security and Data Privacy.** A big challenge in VSE is guaranteeing the system’s security. Vehicle software systems require a defense mechanism against any possible threat. Such a fundamental mechanism requires proper planning through the development lifecycle. The faster and sooner security is handled, a more reliable product is produced [28]. Hence, software security has to be integrated into the initial phases of VSE. This entails implementing secure access control through every component of the system, including ECUs and sensitive data. In an attempt to handle this challenge, AUTOSAR presents security modules Crypto Abstraction Library (CAL) and Crypto Service Manager (CSM) to ensure the authenticity of basic software. To accommodate for communication authentication, AUTOSAR seamlessly integrate a security module, Secure Onboard Communication (SecOC), into AUTOSAR communication stack [8]. Moreover, as safety and security are always linked, ISO 26262 policies contribute to system security by reducing complexity and diminishing the defects that attackers can take advantage of to initiate an attack [20]. J3061 provides guidance and techniques to be adopted in VSE to enhance and ensure the development of a secure software system. The standard includes a list of tools and methods that can facilitate secure development. Moreover, J3061 describes some techniques for Threat Analysis and Risk Assessment, Threat Modeling, and Vulnerability Analysis. However, these techniques are neither comprehensive nor specific for VSE. AUTOSAR,
ISO 26262, and J3061 contribute to security during all the phases of the development lifecycle.

During the implementation phase, software developers can benefit from standards like MISRA, which provide secure development guidelines. Moreover, tools like CHESS and Embedded Coder also aid by providing an automatic code generation tools that follow security policies. This is considered as an essential step for providing a secure product; however, it cannot ensure proper authentication of different entities in the system. During the validation and verification phases, software testers can use tools like the IBM Rational Logiscope, VectorCAST, CANoe, and TPT to validate the code and perform different types of testing like regression testing and integration testing. Identifying defects in the system can prevent the existence of vulnerabilities and therefore improve the overall security of the system. Nevertheless, most of the tools focus on software testing and give less importance to security testing.

Besides projects and tools, many researchers discuss the importance of security in such safety-critical systems. Strandberg et al. [64] present the Start, Predict, Mitigate, and Test (SPMT) methodology to be deployed in VSE. Based on the approach, during the requirement analysis phase, the software engineers are required to identify possible security threats that can be performed against vehicle software systems. However, distinguishing vulnerabilities and potential threats for such a complex system is not an easy job. Other researchers attempt to provide an approach to perform security testing for vehicle software systems. Many researchers proposed fuzzy testing as a good security testing tool to recognize abnormal behavior of ECUs which attackers can take advantage of [57, 58, 62, 63]. Henniger et al. [53], identify
2.2. VEHICLE SECURITY ASSURANCE

and prioritize the requirements needed to secure onboard networks of vehicle systems. Using CAN communication, a malicious message can propagate from one ECU to the other. Huang et al. [61] propose an attack traffic generation tool to test the security of in-vehicle communication. Such tools can enable security testing at an early stage of the development lifecycle to avoid identifying flaws at an advanced stage.

The solutions provided in the literature to enhance security during VSE are oriented toward network security and less toward software security. This industry still requires security assurance and verification methods that can be applied during VSE to guarantee that the vehicle will not reach a state of hazard.

2.2 Vehicle Security Assurance

2.2.1 Vehicle Security Testing Challenges

Security testing is a powerful mechanism to detect and identify the system’s vulnerabilities. In a critical system like a vehicle software system, software testing can prevent life-threatening incidents. However, many challenges make security testing a complex task in the vehicle industry. This section summarizes these challenges as follows.

System Complexity and Size. Vehicle software systems comprise heterogeneous functionalities like safety-related functionalities, infrastructure software, and multimedia systems [12,24]. The vast number of operations a CAV has to perform increases the Source Lines of Code (SLOC) and the hardware devices needed. Security engineers need to ensure stable system operation, yet as the system’s size is relatively large, this job becomes time-consuming. What makes the job of security engineers even more challenging is the complexity of the system. The heterogeneous functions
of vehicle software systems adopt various advancements and technologies like sensors, ECU, network connectivity, artificial intelligence, data analysis, and many other things. It is well studied that complex code is challenging to design and develop, increasing vulnerabilities and security issues [68–70]. Security engineers have to manage the code complexity and size to validate the security and ensure that the system remains safe during its entire operational lifetime.

**Outsourcing.** The development of heterogeneous functionalities embedded within vehicle software systems requires diverse expertise and skills. Hence, OEMs tend to outsource a substantial number of vehicular functionalities [7]. Though this may improve product quality, outsourcing makes security engineers’ jobs more complicated. Software developed by a third party can introduce new threats and vulnerabilities to the system [71]. This is made even harder due to a hierarchical and often complex supply chain. Security engineers must deal with components and certify their security and reliability without knowing their underlying development details or full provenance. Moreover, security testing and system failure rates should be applied to the system as a whole. As many functionalities in the vehicle software system depend on each other, this process might be delayed until all the components are fully integrated, significantly reducing available testing and analysis time.

**Input and Output Fluctuation.** CAVs make reasonable decisions based on the surrounding environment to drive passengers safely to their destination. They utilize sensors, radars, lidars, cameras, other vehicles, and external systems to gather the needed information to understand road conditions, weather conditions, and surrounding traffic [72]. Assessing the set of all possible external environmental data is an intractable problem. Hence, testing and validating vehicle software systems’
behavior is a challenging task. Besides external data, ECUs exchange internal data to trigger specific events. For example, the Powertrain Control Module (PCM) controls the fuel consumption needed to propel the vehicle. The PCM relies on different inputs to determine the correct mixture ratio, including engine temperature, air temperature, and throttle position. In modern vehicles, the PCM also receives internal information from the Adaptive Cruise Control (ACC) ECU to control the speed.

Security engineers have to validate that the system’s catastrophic failure rate falls within an acceptable range, requiring hours of intensive testing that should cover a large number of possibilities [73].

**Test-bed Complexity.** Testing conditions considerably affect the accuracy of the results. Security assurance and validation of the system should be conducted with the same conditions as a real-world scenario. Considering the structure and intricate architecture of a vehicle software system, simulating a real environment becomes an expensive and time-consuming job [26]. Vehicles operate in a wide range of different scenarios, including diverse roads, speeds, visibilities, density, communication patterns, and drivers. Mimicking one scenario might not be enough to ensure a safe and secure system. Many industrial solutions provide OEMs Software in the Loop (SiL) and Hardware in the Loop (HIL) testing simulators that mimic a real environment to evaluate a vehicle software system [48, 74]. Nevertheless, some limitations hinder simulators from becoming a complete solution capable of replacing real-world testing for autonomous vehicles. Simulators are error-prone and may fail to simulate all real-world scenarios comprehensively [75].
2.2. VEHICLE SECURITY ASSURANCE

2.2.2 Security Assurance Methods Utilized in VSE

Safety and security are strongly related disciplines in the vehicle industry. Any security loophole within vehicle software systems can have a drastic effect on the vehicle’s safety, making cybersecurity assurance an indispensable job within VSE. During the security verification and validation phase, security engineers must guarantee that the vehicle system is developed and designed following cybersecurity requirements of vehicle standards like AUTOSAR, ISO 26262, and ISO/SAE 21434 standard. This includes planning, reporting, and, most importantly, a series of security testing to validate the vehicle software system’s protection mechanisms. As the vehicle system incorporates various advancements, including different communication means and hardware devices, ensuring the system’s security throughout its entire lifespan requires adopting several security testing techniques. Some of the testing techniques are automatically incorporated into the development process to identify potential weaknesses promptly, while other techniques require human intervention and acceptance testing after the development phase [15]. This section provides background knowledge about the most common security assurance methods utilized in the vehicle industry, which complement each other: static code analysis, dynamic code analysis, vulnerability scanning, penetration testing, fuzz testing, and risk estimation.

**Static Code Analysis.** Recommended by ISO 26262 [11], and by many other standards, static code analysis is a white-box testing method that dynamically and automatically analyzes the vehicle system’s source code to identify programming errors that may leave the system vulnerable [76]. Imparato et al. [77] examine the potential of existing static analysis tools in identifying loopholes in automotive software components. Their study shows that Bug Finder [78] and Polyspace Code Prover [79]
identify only parts of the code that do not comply with safety and security standards even though these tools are highly performant in other systems. Quality Accelerated (QA) [80] performs better in recognizing software defects that do not comply with the MISRA coding standard developed by the Motor Industry Software Reliability Association [34]. Keul [81] highlights the importance of identifying race conditions in multithreading components of automotive software components. The author proposes a static race condition code analyzer and shows its potential in detecting severe defects.

Static code analysis tools can quickly run early during the development phase to identify a wide range of code defects that weaken the system. They are generally considered worthwhile, especially in MISRA compliance. However, the capabilities of these scanners are limited. They typically produce a high number of false positives [82]. Static code analyzers cannot discover vulnerabilities whose causes are not well understood and modeled in source code (e.g., unchecked inputs and bounds), and thus additional tools are required.

**Dynamic Program Analysis.** Dynamic program analysis examines and monitors a program’s execution to determine incorrect behaviors. It covers all typical software testing forms, including unit, component, integration, and system testing. From a security perspective, it is utilized to look for dangerous conditions such as memory errors, concurrency errors, and crashes. Celik et al. [83] employ program analysis techniques to identify security and privacy issues in the Internet of Things (IoT) systems like automotive systems. In their study, the researchers show the power of dynamic program analysis in discovering vulnerabilities that cannot be identified
with other techniques. Koscher [84] highlights the severity of residing vulnerabilities in automotive systems and stresses dynamic program analysis’s applicability in identifying automotive vulnerabilities quickly and easily. The researcher presents a dynamic analysis tool that simulates inputs and outputs of embedded ECUs in near-real-time. Cabodi et al. [85] propose a dynamic program analysis tool for automotive system security testing that monitors and analyzes CAN message routing and filtering to identify erratic behaviors. Their case study on a gateway ECU shows the tool’s effectiveness in minimizing workload and identifying unusual reactions.

Though dynamic program analysis can expose vulnerabilities that cannot be triggered by static code analysis, it can only cover known software issues. Dynamic program analysis runs against predefined scenarios, limiting the scope of testing.

**Vulnerability Scanning.** Vulnerability scanning validates the resilience of vehicle software systems against known vulnerabilities and security gaps. In other words, such a security assurance method can detect development errors that are not fully traceable, but these errors can be exploited for generating attacks. Vulnerability scanning requires previous knowledge about attacks and security issues in the vehicle industry. In 2015, leading pioneers in the industry formed Automotive Information Sharing and Analysis Center (AUTO-ISAC) [86] to globally collect and analyze emerging cybersecurity risks within the vehicle industry. AUTO-ISAC supplies OEMs with information about identified vulnerabilities by more than 30 automakers, enabling faster vulnerability detection and shared responsibility. Besides industrial forces to improve vulnerability scanning, researchers contribute to this process by consolidating existing attacks. Ring et al. [87] built a database of discovered vulnerabilities to facilitate access during the security validation and verification phase. Similarly,
Sommer et al. [88] examine and classify automotive security attacks to enrich the security testing phase of VSE.

Undoubtedly, vulnerability scanning is crucial to avoid recurring attacks, including attacks discovered during penetration testing, and can also be applied quite early in the development cycle. However, such a security testing tool does not comprehensively evaluate automotive systems. The systems developed by various parties have different weaknesses that vulnerability scanning fails to recognize. Thus, scanning must continually be tailored for each specific system, and additional testing tools are required.

**Penetration Testing.** To validate the resilience of vehicle software systems against malicious behavior, security engineers wear their white hats and attempt to penetrate the system. Penetration testing is the most researched testing technique in the vehicle industry [76]. Koscher et al. [89] experiment with vehicles’ security by conducting several kinds of physical and remote attacks. By simulating replay attacks, the researcher could bypass fundamental network security protections within the vehicle. Cheah et al. [90] employ penetration testing to evaluate the security of vehicles’ Bluetooth interfaces.

Other researchers utilize penetration testing to evaluate in-vehicle communication security. Corbett et al. [91] introduce a testing framework that attempts to bypass the in-vehicle Network Intrusion Detection System (NIDS). Taylor et al. [92] design an anomaly detection framework suited for the CAN bus. The researchers study previous successful attacks to identify common characteristics and simulate a range of new attacks. Huang et al. [61] validate the CAN defense mechanism by proposing a tool that automatically injects attack packets into the CAN bus.
Researchers identified some loopholes within vehicles by conducting penetration testing, and this method is the most potent to validate vehicular network security. While penetration testing generates the most significant and meaningful results, it is usually the most time-consuming, is by definition incomplete, and requires deep domain expertise. Automation of known attacks is always a vital aspect of a functional penetration testing strategy in Vehicle Software Engineering (VSE). Good coverage of well-known issues and attacks, as well as the most likely and significant attacks, can be reasonably well covered through penetration testing. However, it is not enough to conduct penetration testing to ensure the resilience of vehicle software systems.

**Fuzz Testing.** Fuzz testing is a robust testing technique that validates the system behavior against generated arbitrary inputs to identify unexpected behaviors that attackers can use to initiate attacks [93]. Three different testing methodologies can be employed: white-box, black-box, and grey-box fuzz testing.

Researchers in the vehicle industry focus on black-box fuzz testing and avoid adopting white-box fuzz testing. Though white-box testing can comprehensively evaluate the system, considering the system’s complexity and size, deploying such a mechanism in the vehicle industry is time-consuming and requires significant effort.

Oka et al. [58,94] consider black-box fuzz testing as one of the most powerful tools to discover vulnerabilities within vehicle software systems. They show its efficiency by performing fuzz testing on an Engine ECU and Gateway ECU. The researchers successfully identify corrupted Pulse Width Modulation (PWM) frequencies by monitoring engine ECU response to fuzz and random messages.

In other research work, Oka et al. [62] highlight the challenges of validating and testing a complicated and broad system like the vehicle software system. Initiating
2.2. VEHICLE SECURITY ASSURANCE

the testing after the completion of the system can cause delays in vehicle production. Oka et al. find that fuzzing allows the testing to start at an earlier stage in the development process. Random inputs can replace the required inputs needed to verify the developed functionalities.

Fowler et al. [63,95–97] propose slightly modifying valid Controller Area Network (CAN) messages to identify security issues in ECUs. They perform black-box fuzzing on a vehicle’s display ECU and show the benefit of fuzzing automotive inputs to identify bugs and weaknesses in the vehicle software system.

Vinzenz et al. [98] attempt to manage the size and complexity of automotive systems by specifying the fuzzing time of different components. They employ Threat Analysis and Risk Assessment (TARA) techniques to estimate the risk and identify the testing time accordingly. Identifying security loopholes early in the development lifecycle enables proper mitigation of vulnerabilities before production.

Patki et al. [99] present Unified Diagnostics Services (UDS) fuzzer that uses valid input messages and performs mutation for specific fields to generate invalid inputs. Werquin et al. [100] focus on automating the detection of system malformed responses to fuzz messages. The authors discuss the use of bug Oracle functions to establish different fuzzing strategies and develop a sensor harness to detect unexpected ECU responses automatically.

Despite black-box fuzz testing’s ability to manage the system’s complexity, outsourcing, and input and output fluctuation challenges, conducting blind testing for a critical system may be risky. Black-box testing cannot guarantee good coverage and a thorough evaluation of the system. In addition, arbitrary test cases may not pass initial input validation requirements prohibiting the testing from expanding to
the system’s core. This is challenging in VSE, where strong input validation is a core requirement and is often validated through static analysis. Adopting such a testing methodology in the vehicle industry may not ensure a risk-free lifespan.

To overcome black-box fuzzing limitations, Radu et al. [101] use static analysis to manage the fuzzing of automotive ECUs. They introduce EffCAN, a tool for ECU firmware fuzzing via CAN messages that is the most potent when the source code is not available. EffCAN requires disassembling ECUs firmware to build control flow graphs that help guide the fuzz testing. However, disassembling ECUs firmware is a challenging job. Separating the hardware-specific drivers and code regions from software components is a daunting task that requires manual analysis. Hence, a security tester is required to have a solid knowledge of the underlying ECU hardware architectures.

Risk Estimation. SAE J3601 [9] recommends assessing security threats in the automotive industry to identify possible threats. However, it does not identify a specific Threat Analysis and Risk Assessment (TARA) method that can best identify the automotive industry’s security risks [102]. The E-safety Vehicle Intrusion Protected Applications (EVITA) threat and risk model [18] is considered one of the potent risk assessment models in the automotive industry. The model focuses on identifying all possible attacks against a specific target. However, the sets of attacks and targets within autonomous vehicle systems are practically large, making risk assessment a time-consuming and challenging job. ISO/SAE 21434 [4] proposes a generic risk assessment process that involves vulnerability analysis, which is estimated based on previously identified vulnerabilities. However, historical data is not sufficient to identify the evolving vulnerabilities of vehicles.
2.3. OTHER FUZZ TESTING TECHNIQUES

Burton et al. [103] stress the importance of identifying intentional third party hazards to enhance vehicle safety. The researchers suggest enhancing safety standards to include the categorization of malicious hazards that can affect safety. Similarly, Macher et al. [104] highlight the need for threat and risk assessment techniques for the automotive domain. The researchers propose an approach to classify cybersecurity threats and merge it with ISO 26262 safety Hazard Analysis and Risk Assessment (HARA) framework. However, it is not enough to address vehicle security from a safety perspective only. Security risks have several impacts other than safety, including operational and financial impacts.

Islam et al. [105] introduce a risk assessment framework that aims to identify security requirements for automotive systems. Though the researchers propose a solid framework, their approach operates based on the system’s data flows that can be difficult to obtain in the automotive industry.

2.3 Other Fuzz Testing Techniques

This section briefly reviews existing grey-box and hybrid fuzz testing techniques that validate operating systems and software application security.

Other Grey-box Fuzz Testing Techniques. Recently, grey-box fuzzing has become a popular security testing tool [106]. The most notable grey-box fuzz testing tool is American Fuzzy Lop (AFL) [107], with which we compare our results to in the evaluation section of Chapters 5 and 6. AFL collects coverage information to identify test cases that expand code coverage. Various strategies are introduced to enhance the coverage and performance of AFL [108–110].
However, none of these techniques are applied to validate the security of an automotive component. Researchers design their approaches to target desktop-based programs. Such grey-box fuzzing techniques do not target CAV security testing challenges, particularly the system’s complexity and size. They spend hours of testing, focusing entirely on expanding code coverage. With strict time budgeting and a complex system, testing is better focused on weak components that may expose the system to attacks.

Zhang et al. [111] attempt to rank the seeds generated by AFL, but their test case prioritization does not guide the testing in a specific direction. Bohme et al. [110] introduce Directed Greybox Fuzzing (DGF) that focuses on testing targets specified by the user. They implement their approach on top of AFL and name it AFLGo. They achieve their goal by eliminating the test cases that are far from the targets. They calculate the minimum distance between the system nodes to identify close seeds. Minimum distance forms a significant limitation as it eliminates crucial paths in the system that can hold bugs. DGF depends on the prior knowledge of vulnerable areas, which can be guided by threat and risk assessment but cannot be complete. Moreover, when testing a newly developed system, it is essential to examine the whole system rather than just specific functions.

**Other Hybrid Fuzz Testing Techniques.** Researchers combine fuzzing and symbolic execution into hybrid testing techniques to enhance the evaluation coverage and expand the vulnerability exposure. Symbolic execution explores multiple paths of the system without acquiring concrete inputs. Inputs and variables are instead symbolically presented and analyzed. Symbolic exploration resorts to constraint solvers to form concrete inputs that satisfy conditional branches and trigger vulnerabilities [112].
For example, Godefroid et al. [113] design a Directed Automated Random Testing (DART) tool that reviews the behavior of a target program when executed with random input and gathers branch constraints. DART then utilizes symbolic execution to resolve the constraints and drive the testing towards undiscovered paths. However, DART inherits all the drawbacks of symbolic execution due to its excessive reliance on symbolic exploration. Majumdar et al. [114] interweave random fuzz testing with concolic execution. Concolic testing drives the execution using concrete inputs and simultaneously symbolically examines the execution path, enabling the generation of new inputs that resolve input-specific conditions [112]. Nevertheless, the capabilities of random fuzz testing are limited and often fail on simple checks in automotive systems, putting more effort on the concolic exploration.

Ganesh et al. [115] introduce BuzzFuzz, a taint-based directed white-box fuzzing tool that executes the target program with valid inputs and records taint information. The security tester chooses a specific target, and the taint information is analyzed to construct new inputs that traverse the target. Similarly, Wang et al. [116] design TaintScope, a checksum-aware fuzzing tool that employs branch profiling techniques to locate fields in an input that allow the execution of preconditional statements preventing test cases from getting stuck. However, BuzzFuzz and TaintScope are time-consuming and require manual intervention, making them inefficient for complex and large systems like automotive systems.

Haller et al. [117] design Dowser a guided fuzzer that combines random fuzzing and symbolic execution to discover buffer overflow vulnerabilities. Dowser focuses the testing on specific code regions that are more likely to include buffer overflow vulnerabilities rather than performing a comprehensive system evaluation. Pak [118]
presents a hybrid testing tool that starts with symbolic execution to identify inputs that pass preconditional checks and then switches to random fuzzing to evaluate the remaining code regions. Nevertheless, the initial dependence on symbolic execution weakens the validation process if the Satisfiability Modulo Theories (SMT) solver fails to find satisfying inputs.

Driller \[119\] incorporates grey-box fuzz testing and selective symbolic execution. Driller starts by evaluating the system with the grey-box fuzzer AFL and switch to concolic testing when AFL runs few test cases without identifying a new transition. Concolic execution identifies few valid inputs and passes the testing again to the AFL fuzzer. Though Driller is a potent hybrid testing approach, its effectiveness is reduced when applied to automotive systems. Considering the size and complexity of the system, frequent interweaving between fuzz testing and concolic testing expands the testing time. Incremental solving of constraints leads to repetitive work. Depending on the test case that drives the symbolic execution, Driller might waste much time to solve constraints that are already explored. Moreover, the symbolic execution of multiple paths of an automotive component requires enormous memory consumption.

To limit the use of symbolic execution, Yun et al. \[120\] design QSYM, a practical concolic execution engine that performs instruction-level symbolic emulation. QYSM does not symbolically evaluate each instruction in an execution path, which diminishes the memory consumption of symbolic exploration. Nevertheless, skipping symbolic execution for instructions in an arbitrary fashion can affect the soundness of testing. Moreover, such a symbolic engine has to manage a large and complex instruction set of CPUs which increases the complexity of the testing.
2.4 Summary

Software systems are the key enabler of embedding modern functionalities in vehicles. However, developing such systems is a very complex task given the unique characteristics of Connected Autonomous Vehicles (CAVs).

We further identify the unique challenges of VSE, which entail software integration, compatibility and code reusability, safety and reliability assurance, and software security and data privacy. We investigate the effectiveness of current practical solutions, including standards, tools, languages, and other research efforts in addressing the defined challenges. In this respect, this chapter provides guidelines for researchers to understand the evolution, trends, and current practices in this research area. Based on our mapping and analysis, we notice that, in general, communication diversity, as well as maintenance and error recovery, did not receive proper attention in the existing solutions. The focus of the solutions is primarily on addressing three challenges: a) testing complexity, b) safety and reliability assurance, and c) software security and data privacy. However, they do not comprehensively overcome the challenges.

Among the standards and processes, the AUTomotive Open System ARchitecture (AUTOSAR) standard is becoming more popular in the automotive industry as it facilitates collaboration between different suppliers. Nevertheless, many loopholes hinder this standard from becoming the utmost adopted solution for VSE. Most importantly, a series of migration phases are required to fully adopt AUTOSAR principles at the cost of money, time, and effort. The burden is further increased due to the complexity of this standard. Hence, it requires more research to apply and automate the practices of this standard.

Most of the tools and projects focus on providing a solution to assure the safety and
reliability of the system, but only a few tools offer a fully effective and comprehensive solution. Many researchers advocate for testing practices that can help mitigate challenges relevant to software security and data privacy. Nevertheless, each of these testing techniques has shortcomings that hinder them from becoming an efficient and reliable security testing tool. Given the continuous sophistication of attacks against vehicles, there is a pressing demand for new solutions that are applicable not only during the early phases of the development but also beyond the operation to span the full lifecycle of vehicles.
Chapter 3

Engineering of Vehicle Software Systems

Vehicle software systems have many characteristics that make them different from other software systems [5–7]. Hence, it is not ideal to embrace existing software engineering models. Given the incentive of Original Equipment Manufacturers (OEMs) to reduce software development costs and gain high profits [121], there is a pressing demand that they adequately address the unique challenges originating in the software engineering of Connected Autonomous Vehicles (CAVs). This can be realized by adopting a Vehicle Software Engineering (VSE) process.

The importance of VSE did not receive the required level of attention in the research community. The majority of existing research work [5,24] studied the challenges in VSE from a model-based design perspective. For example, Pretschner et al. [24] explore the benefits and challenges of deploying a model-based development process for automotive software to achieve abstraction in terms of requirement engineering and project management. Despite the benefits claimed by the model-driven engineering approaches [5,24], they cannot mitigate all the unique difficulties facing VSE. Bertonlino et al. [28] consider a general high-level software engineering process.

\(^1\)In this chapter, we use the terms “model” and “process” interchangeably.
not necessarily adopted for CAVs and focus only on security as the main challenge in VSE.

The architecture of CAVs also introduces unique challenges for automotive security development and operation that traditional security lifecycles are insufficient to manage. Many Secure Software Development Lifecycles (SSDLCs) are in use, but Microsoft Security Development Lifecycle (MSDL) [122] and Secure Software Development Framework (SSDF) [123] are the most popular ones [124].

However, deploying MSDL or SSDF in the automotive industry leaves several challenges unhandled. CAVs are Cyber-Physical Systems (CPSs) that rely on different advancements with a high level of automation and Internet of Things (IoT) connectivity that originate growing unique challenges for cybersecurity. Managing the complexity and size of automotive systems requires security engineers to prioritize security testing, which is not usually enforced by traditional security development lifecycles. Moreover, automotive systems involve heterogeneous subsystems (i.e., body software and safety software) that are exposed to a different level of threat. Hence, different security testing tools might be utilized for each subsystem depending on the assurance level required. MSDL and SSDF employ the same security testing tools on the whole system. Most importantly, traditional SSDLCs do not accommodate the long development lifecycle of automotive systems. Over an extended development lifecycle, employed security practices might become weaker, requiring preplanning and regular validation throughout the development lifecycle.

This chapter performs a deep analysis of the benefits and limitations of each software engineering process to help in identifying their capabilities that best fulfill the characteristics of CAV environment. In this sense, this analysis enables vehicle OEMs
3.1. VEHICLE SOFTWARE ENGINEERING (VSE)

We define VSE as a systematic development process that utilizes software engineering principles and standards to handle the unique challenges of vehicle software systems including all of its communication means while ensuring vehicle safety and security.

and software providers to gain a clear and comprehensive insight into VSE models. Hence, they can assess and choose a VSE model that can meet their challenging needs. The second part of this chapter presents a Secure Vehicle Software Engineering (SVSE) lifecycle that ensures security-by-design and incorporates security activities that appropriately address the development security challenges of CAVs. Moreover, the SVSE lifecycle ensures compliance with international security standards by embedding security considerations that satisfy these standards’ requirements.

We begin this chapter by analyzing software engineering processes that are practiced in the automotive industry. We perform a profound analysis of the benefits and limitations of each engineering process to help identify their capabilities that best fulfill the characteristics and challenges of Connected Autonomous Vehicles (CAVs). Next, we introduce the Secure Vehicle Software Engineering (SVSE) lifecycle. First, we review the vehicle development challenges. Then, we present the SVSE lifecycle phases and discuss its security activities, detailing how they manage the development challenges and meet the security standard requirements.

3.1 Vehicle Software Engineering (VSE)

We define VSE as a systematic development process that utilizes software engineering principles and standards to handle the unique challenges of vehicle software systems including all of its communication means while ensuring vehicle safety and security.
As depicted in Fig 3.1, the vehicle engineering lifecycle is a compound process that involves three cycles: 1) system engineering, 2) software development, and 3) operation. In the system engineering cycle, automotive Original Equipment Manufacturers (OEMs) typically specify the functionality requirements and accordingly choose the components (e.g., Electronic Control Units (ECUs) and infotainment system hardware) from the supply chains. The management of these components is essential due to its direct impact on the subsequent cycle of software development. OEMs commonly follow AUTomotive Open System ARCHitecture (AUTOSAR), which is currently an international development standard of vehicle manufacturers [8]. AUTOSAR is considered one of the most popular standards that aid in the first lifecycle of vehicle manufacturing [28]. Then, the software providers take charge of software development during the second lifecycle. They build about a hundred million lines of code that can integrate with thousands of the chosen components (i.e., ECUs) needed for assembling the desired vehicles. To manage the complexity of the software, different software engineering models branched from the Verification and Validation Model (V-Model) [125] and the Agile Model [126] can be adopted in the vehicle industry. After vehicles are manufactured, the last lifecycle maintains the operation of the products.

In this section, we mainly focus on the second cycle which involves the vehicle software development. The unique features of vehicle software systems increase both complexity and connectivity. Despite the importance of following a standardized engineering process to cope with these unique features efficiently and cost-effectively, still there is a lack of guidelines to understand the common strengths and weaknesses of the existing engineering models. To overcome this limitation, we further study the
existing models.

In this section, we compare and contrast two widely adopted software engineering models: Verification and Validation Model (V-Model) and Agile Model. We particularly outline the major strengths and limitations of adopting these models in the vehicle industry to help OEMs and software developers in choosing an appropriate model.

3.1.1 V-Model

The V-Model is currently the most adopted engineering model in the vehicle industry [125]. It is an extension of the Waterfall model [127] with one major difference; it does not have a linear axis. In fact, after the coding phase, it turns back upwards for validation and the outcome of each phase is validated before proceeding to the next phase. The V-model has many versions, e.g., the traditional V-Model, Multiple V-Model, and incremental V-Model (inc-V). All these versions are applicable for vehicle software development [125].

Strengths of the V-Model in the Vehicles Industry

Early Verification. The vital goal of the V-Model is to create a balance for workers and ensure verification and validation before proceeding to the next phase. Thus, software requirements are well verified before moving to the design phase [128]. Based on our findings, in safety-critical systems like vehicle software systems, early verification becomes more important to guarantee that all ambiguities are identified and appropriately handled.

Powerful Function Integration. The V-Model is considered one of the software engineering models that supports tailoring and outsourcing [129]. This is particularly
important in the vehicle industry as most of the functionalities are outsourced [5]. Moreover, in our opinion, this brings another advantage, since such a separation can help in supporting heterogeneous subsystems that exist within vehicle systems. By following the V-Model, OEMs can outsource diverse functionalities to expert suppliers.

*Simple Model.* As vehicle software systems are becoming more complex, it is better to use a simple method that does not require intensive learning. Each phase of the V-model has a specific deliverable and review process [127]. This makes work management simpler and more efficient.

**Limitations of the V-Model in the Vehicles Industry**

*Reduced Flexibility.* One of the major drawbacks reported in the literature concerning the V-Model is its inflexibility and rigid phases [127]. OEMs specify vehicle features at an early stage without considering the practicality of these features in terms of component interaction and development [130]. For this reason, some stipulations might be changed after finalizing the requirement specification phase. Following the V-Model, this requires reverting to the previous phases. Thus, such a change can incur delays in the project and jeopardize the project deadline [125].

*Time Consuming.* For each phase of the V-Model, software engineers are required to prepare documentations. Though this can ease the work organization, it consumes a significant amount of time. Considering the competition in the vehicle industry and strict deadlines, this task can adversely affect project management.
3.1.2 Agile Model

The Agile software engineering model promotes evolutionary development, early submission, and support for rapid changes. In many projects, software engineers decide to follow Agile practices to handle unexpected changes in customers’ requirements [131]. Today, vehicle software engineers are still studying the adoption of Agile practices in the automotive industry [126, 132–134].

Strengths of the Agile Model in the Vehicles Industry

Simple Model. Unlike other software engineering models, the Agile model does not have rigid phases. In contrast, it is well known for its flexibility and simplified policies, which increases productivity [127].

Flexible Change of Requirements. Agile practices advertise maximum customer satisfaction by handling requirement changes easily at any phase. In the automotive industry, there are always new innovative functionalities that are added to vehicles. Thus, even if experts set the requirements, the lack of experience entails the need to go over the requirements many times during the development lifecycle [134].

Time Efficient. Agile development promises quick project progress [127]. OEMs can take advantage of the shorter release cycle and reduced documentation to focus more on enhancing the product.

Limitations of the Agile Model in the Vehicles Industry

Difficult Organizational Structure Transformation. Migrating from one software engineering model to another requires many policies and organizational changes [133]. Moreover, in the vehicle industry, experts assess that decision-makers in this field often lack knowledge on how to implement the Agile model [132]. This prevents them from migrating to the Agile model.
Adverse Effect on Quality of Work. Ensuring safety and security is vital for vehicle software development. A quick development process may affect these qualities adversely \cite{132,133}. Thus, deploying a product in a sensitive environment requires intense testing and validation mechanism, which entails longer development time.

Limited Code Reusability. The Agile model might not be effective in managing code reusability for vehicle software systems \cite{133}. Ensuring code reusability will require deep planning from the initial phase of the development process. This planning should not target rapid development as in the Agile model but should rather focus on building strategies to handle code reusability and save time in the long run.

Inconvenient Work Collaboration. Agile development requires high coordination and communication between the entities involved in a project. In the vehicle industry, it is difficult to organize frequent communications between different parties \cite{132}. As vehicle software systems are complex systems that involve a large number of employees, organizing regular meetings and discussions with employees is a time-consuming and complicated task that might lead to wrong decisions \cite{135}.

Weak Function Integration. Vehicle software integration is an important step in building a vehicle software system. Software integration is generally addressed during the design phase \cite{136}. It means that any problem in integration will appear late in the software engineering process. Even though regular checks of integration is recommended to mitigate these problems, they are less supported by Agile development \cite{127}. 
3.1. VEHICLE SOFTWARE ENGINEERING (VSE)

Table 3.1: Advantages of the V-Model and the Agile model (√ shows the presence of an advantage, * indicates that a model has an advantage more than the other model)

<table>
<thead>
<tr>
<th>Advantages</th>
<th>V-Model</th>
<th>Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early Verification</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Powerful Function Integration</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Simple Model</td>
<td>✓</td>
<td>✓*</td>
</tr>
<tr>
<td>Flexible Change of Requirements</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Time Efficient</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

3.1.3 Comparison between V-Model and Agile Practices

Tables 3.1 and 3.2 outline the advantages and disadvantages of the reviewed VSE models. Both models are known for being simple. However, the Agile model is designed to be simpler with minimal complexity in development procedures. Vehicle software systems require proper handling of software integration, which is not strongly supported by the Agile model. For this reason, many researchers consider the V-Model as the most applicable software engineering model for vehicle software systems [125,130,137]. Moreover, the Agile model has many disadvantages that are considered critical in VSE. The presented drawbacks of adopting the Agile model refrain software engineers from handling the unique characteristics of vehicle software systems. For this reason, selecting the Agile model in such a complex system requires further investigation and studies.

3.1.4 Mapping of Challenges to VSE Phases

This subsection maps the challenges defined in Chapter 2 in light of VSE phases and further relate them with the VSE models.

Mapping of Challenges to VSE Phases. Table 3.3 depicts the mapping of the aforesaid challenges (Chapter 2) to the VSE phases. We correlate the main phases
Table 3.2: Disadvantages of the V-Model and the Agile model (✗ shows the presence of a disadvantage)

<table>
<thead>
<tr>
<th>Disadvantages</th>
<th>V-Model</th>
<th>Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduced Flexibility</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Time Consuming</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Difficult Organizational Structure Transformation</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Adverse Effect on Quality of Work</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Limited Code Reusability</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Inconvenient Work Collaboration</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Weak Function Integration</td>
<td></td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 3.3: Mapping of challenges to phases

<table>
<thead>
<tr>
<th>Challenges</th>
<th>Requirement</th>
<th>Design</th>
<th>Implementation</th>
<th>System Testing</th>
<th>Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Integration</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Communication Diversity</td>
<td>✗</td>
<td></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Compatibility and Code Reusability</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Test-bed for Vehicle</td>
<td>✗</td>
<td></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Software Testing</td>
<td></td>
<td></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Cross-ECU Testing</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Input and Output Validation</td>
<td>✗</td>
<td></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Maintenance and Error Recovery</td>
<td>✗</td>
<td></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Safety and Reliability Assurance</td>
<td>✗</td>
<td></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Software Security and Data Privacy</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

throughout the software engineering process (Requirement Analysis, Design, Implementation, Testing, and Operation and Maintenance) to the challenges facing vehicle software systems. This mapping can assist system engineers during the project management by providing prior knowledge about the challenges that have to be handled in each phase of the development lifecycle.

**Mapping of Challenges to VSE Models.** Table 3.4 addresses the relationship between VSE models and the challenges. In this comparison, we identify the model
Table 3.4: Mapping of challenges to VSE models (✓ shows that a model aids in handling a challenge, * indicates that a model aids more than the other model)

<table>
<thead>
<tr>
<th>Challenges</th>
<th>V-Model</th>
<th>Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Integration</td>
<td>✓ *</td>
<td>✓</td>
</tr>
<tr>
<td>Communication Diversity</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Compatibility and Code Reusability</td>
<td>✓ *</td>
<td>✓</td>
</tr>
<tr>
<td>Test-bed for Vehicle Software Testing</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cross-ECU Testing</td>
<td>✓ *</td>
<td>✓</td>
</tr>
<tr>
<td>Input and Output Validation</td>
<td>✓ *</td>
<td>✓</td>
</tr>
<tr>
<td>Safety and Reliability Assurance</td>
<td>✓ *</td>
<td>✓</td>
</tr>
<tr>
<td>Software Security and Data Privacy</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

that can better support software engineers in managing a challenge. In principle, both VSE models can be utilized to manage the challenges. However, one model may better address a particular challenge than the other one. For example, the structure of the Verification and Validation Model (V-Model) supports software integration and promises well-managed outsourcing. Moreover, due to the extensive verification and validation required by the V-Model, input and output validation challenge becomes more manageable compared to the Agile model. The V-Model phases provide firm and organized processes that better support the safety management challenge. Table 3.4 shows that the V-Model is capable of handling the challenges more efficiently than the Agile Model.

### 3.2 Secure Vehicle Software Engineering (SVSE)

The Secure Vehicle Software Engineering (SVSE) lifecycle introduced in this chapter is designed following Automotive Software Performance Improvement and Capability Determination (ASPICE) V-Model [35] phases, the most used SDLC in the automotive industry. Hence, the SVSE lifecycle allows the automotive industry to
3.2. SECURE VEHICLE SOFTWARE ENGINEERING (SVSE)

conduct smooth and simultaneous software, safety, and security development lifecycles.

Complying with requirements of international standards like the Road Vehicles Cybersecurity Engineering ISO/SAE 21434 [4], United Nations Economic Commission for Europe (UNECE) World Forum for the Harmonization of Vehicle Regulations (WP.29) [138], MISRA C:2012 [34], and the AUTomotive Open System ARchitecture (AUTOSAR) [8] implies following different requirements and principles during Vehicle Software Engineering (VSE) [12]. The SVSE lifecycle encompasses security activities that manage the requirement of software security standards, asserting manageability and endurance. It also includes security activities that mitigating the automotive cybersecurity development challenges.

3.2.1 Vehicle Security Development Challenges

In what follows, we identify the challenges that face security engineering during the development of automotive systems. Though other industries encounter some of these challenges, security engineers have to manage all these challenges during the development of automotive systems.

**Complex and Large System.** Being one of the largest systems, automotive systems currently embed over 100 million Lines of Code (LOC) [139]. The complexity and size of the system pose challenges to the whole security development process and increases the risk of vulnerabilities [20].

**Various Signals.** Connectivity and cutting-edge technologies like sensors, cameras, and radars enabled vehicles to gather information. However, the wide range of incoming and outgoing signals pose challenges to the SSDLC [140]. Security engineers
are responsible for mimicking the signals and validating the system’s resilience with all possible scenarios.

**Outsourcing.** Outsourcing increases the complexity of SSDLC. This is exceptionally risky in the automotive industry due to the complex suppliers’ hierarchy. Security engineers must manage the outsourced software components and ensure a certain level of security without knowing the details of the development.

**Open-source Code.** Like many other systems, automotive systems utilize open-source code, including drivers, libraries, applications, and Operating Systems (OSs). Nevertheless, open-source code empowers attackers with knowledge. Attackers have access to the source code, allowing them to identify vulnerabilities and possible entry points. The automotive industry must utilize a security development process that carefully assesses open-source software, makes sure that it is up to date, and verifies that it is free from known vulnerabilities.

**Security Decay.** CAV development spans several years, during which security risks evolve. Hence, employed cybersecurity practices during the initial phases of the SSDLC might become ineffective by the end of the development cycle. Automotive SSDLC must be potent in coping with fast-evolving cyber risks by assuring that security requirements and considerations can defend CAVs from new cyberattacks. Moreover, not all software components in automotive systems are newly built. Some components are reused from old versions. Legacy code is inherited without a proper understanding of its design. Therefore, legacy code should be well evaluated during SSDLC to diminish the existence of legacy vulnerabilities. Security decay does not only arise during the development stage but also throughout the operation phase. During vehicles’ long operation phase, technology advances and attacker capabilities
increase, leaving the security of the safety-critical system unreliable. The automotive industry must manage security decay and assure resilient vehicles.

**Maintenance and Incident Response Monitoring.** During the operation phase, it is necessary to consider cybersecurity for CAVs. During the maintenance process, software updates might hold new vulnerabilities. Thus, any SSDLC must span security consideration to the operation phase to assure a reliable software update and fixes. Moreover, security engineers are expected to respond to cybersecurity incidents promptly. Hence, cybersecurity incident response must be well planned to ensure that all the means and tools needed to mitigate an attack are available.

### 3.2.2 Lifecycle Overview

The Secure Vehicle Software Engineering (SVSE) lifecycle, shown in Fig. 3.2, is designed to identify and mitigate software security risks during the development of automotive software systems. The SVSE spans nine phases (P₁ to P₉) divided into four stages: planning, development, testing, and operation.
Cybersecurity should be envisioned before starting the development stage to guarantee the integration of security fundamentals into the automotive software. The Cybersecurity Planning phase prepares for security development by identifying the keystones that can assure a resilient automotive system.

Once the foundations are established, the development stage starts. The Cybersecurity Requirement Analysis phase considers the complexity of automotive systems that require special security consideration to manage it properly. During the next phase, Cybersecurity Design, the system architecture is examined thoroughly to construct proper measures limiting system weaknesses. As automotive systems are safety-critical systems assuring cybersecurity during vehicle operation is vital. The Cybersecurity Assurance Planning phase guarantees that manufacturers are ready to take proper and prompt actions to mitigate cyber incidents. This phase should start after the Cybersecurity Design phase to confirm that the incident response planning abides with the system design. During the Cybersecurity Implementation phase, security engineers monitor the development of the automotive components to confirm that they comply with secure coding practices.

Three phases of the SVSE lifecycle are dedicated to security verification, which run sequentially: Cybersecurity Component Testing, Cybersecurity Integration Testing, and Cybersecurity Resilience Verification. According to the testing results, security engineers might need to revert to the initial phase to manage identified failures. During the Cybersecurity Component Testing and Cybersecurity Integration Testing phases, the SVSE lifecycle employs different security testing methods to validate the heterogeneous automotive subsystems.

The last stage of the SVSE lifecycle runs continuously during the operation of
vehicles to maintain cybersecurity. The Cybersecurity Maintenance phase monitors cyberattacks and takes proper actions to mitigate their existence, keeping vehicles safe and secure. Cybersecurity Decay Review is a core element of the SVSE lifecycle. It requires security engineers to review security practices and procedures several times during the SVSE lifecycle to identify security measures that become weaker over the long development lifespan of vehicles. Cybersecurity Decay Review does not require the development to halt; it can concurrently run by a different team to avoid delays in the delivery. Chapter 7 of this thesis offers an Autonomous Vehicle Security Decay Assessment (AVSDA) framework that can be used for this activity. The framework determines the security risk to identify security decay efficiently.

3.2.3 Lifecycle Security Activities

The SVSE lifecycle identifies specific security activities and considerations that security engineers must follow during each phase. The SVSE lifecycle incorporates security activities that help meet the requirements of automotive standards. We specifically choose four international standards: ISO/SAE 21434, AUTOSAR, MISRA C:2012, and UNECE WP.29 as they discuss cybersecurity at the software level of automotive systems. Moreover, the security activities help manage all the automotive-specific security development challenges. We discuss the details and the mapping of each security activity summarized in Fig. 3.3.
3.2. SECURE VEHICLE SOFTWARE ENGINEERING (SVSE)

Figure 3.3: Security activities of the SVSE lifecycle mapped to the challenges and the security standards

Cybersecurity Planning (P₁)

Identify Cybersecurity Objectives (P₁A₁). Following ISO/SAE 21434, the cybersecurity objectives should be explicitly determined to lead the security engineering process. 

²Pₓ symbolizes a phase x in the SVSE lifecycle, and Aᵧ represents security activity y within phase x. The same labeling is presented in Fig. 3.3.
3.2. SECURE VEHICLE SOFTWARE ENGINEERING (SVSE)

Within the context of automotive software systems, cybersecurity entails protecting Electronic Control Unit (ECU) software, in-vehicle infotainment software, and any underlying data. The objectives should clearly state the required level of security needed across all the components of automotive systems while assuring conflict-free goals.

*Threat Analysis and Risk Assessment (P1A2).* In an industry that is advancing quickly and relying on technologies that expand the attack surface, cybersecurity risks increase. ISO/SAE 21434 and UNECE WP.29 require performing threat and risk assessment several times during the development. It is a vital task in the planning stage that helps mitigate the complexity and system size challenges by notifying security engineers about possible threat scenarios to prioritize, plan, and design measures that mitigate risks. This activity involves several steps, including asset identification, threat analysis, impact assessment, attack path identification, and risk assessment. As the development details are not determined yet, some steps might not be comprehensive at this stage. For example, general information can be utilized (e.g., remote access or physical access required) to identify attack feasibilities. More extensive analysis is conducted during the development stage.

*Identify Cybersecurity Assurance Levels (P1A3).* The heterogeneous automotive components require different Cybersecurity Assurance Levels (CALs). ISO/SAE 21434 introduces CAL, which is determined according to the criticality and the risk of the asset to be protected. Security engineers can rely on the threat and risk analysis results to define the CALs of different components. The CAL drives subsequent security activities. For example, the higher the CAL, the more strict security requirements are needed. Similarly, CAL influences the extent and depth of the security testing.
CAL classification facilitates communication among different parties involved in the development and assures that various automotive system components are receiving the required security attention.

**Cybersecurity Requirements Analysis (P₂)**

*Define Cybersecurity Requirements (P₂A₁)*. Cybersecurity requirements are established based on the cybersecurity objectives, threat and risk assessment results, and CAL values. The requirements needed to mitigate risks must be defined in detail, including information on entities that require security, the source of the threat, and the level of expected security. ISO/SAE 21434 recommends that cybersecurity requirements should be specified following the functional requirements while incorporating security principles covering integrity, confidentiality, and availability. A cybersecurity Requirement Traceability Matrix (RTM) should be prepared, illustrating the relationship between the requirements and other phases (e.g., testing phases). RTM helps prove that the requirements are fulfilled in subsequent phases.

*Prioritize Cybersecurity Requirements (P₂A₂)*. All the specified requirements are expected to be employed, but this activity gives more importance to essential ones. Advised by ISO/SAE 21434, cybersecurity requirement prioritization can help manage automotive systems’ complexity by giving higher priority and special consideration to fundamental requirements that assure security. Cybersecurity requirements can be systematically prioritized based on threat analysis and risk assessment.

*Create Supplier Cybersecurity Assurance Requirements (P₂A₃)*. As many automotive components are outsourced, the SVSE lifecycle adopts proper actions that support supply chain security. The automotive industry should communicate security needs
and requirements to suppliers. These requirements should adhere to the defined cybersecurity objectives and specified requirements. To control the security and gain confidence in outsourced components, security engineers should define cybersecurity assurances that prove supplier compliance to requirements (e.g., reports, threat assessment and risk analysis reports, external audit, and security testing results).

**Cybersecurity Design (P₃)**

*Review Cybersecurity Requirements (P₃A₁).* Before starting with cybersecurity design, cybersecurity requirements must be reviewed to identify missing requirements and detect security flaws. The purpose of this step is to determine gaps at an early stage of the development cycle, preventing delivery delays and assuring up to date security practices.

*Design Measures to Ensure Cybersecurity (P₃A₂).* This security activity should simultaneously run with the software design phase to ensure that automotive software components are well security engineered. Mandated by ISO/SAE 21434 and UNECE WP.29, security engineers should design security features that satisfy the defined cybersecurity requirements, defend against identified threats, and diminish security risks. This includes various security methods like authentication, authorization, data encryption, and logging.

*Design Measures to Detect Cyberattacks (P₃A₃).* As automotive systems operate in an environment that is exposed to a plethora of evolving cyberattacks, designing measures to ensure cybersecurity is insufficient to maintain the safety of vehicles. Required by UNECE WP.29, security engineers should design measures that enable automotive systems to detect cyberattacks, including physical and remote attacks.
3.2. SECURE VEHICLE SOFTWARE ENGINEERING (SVSE) 66

Software Security Design Review ($P_3A_4$). To avoid late detection of vulnerabilities, security engineers should review the detailed software design. The software architecture and components’ design should be examined to ensure compliance with the defined cybersecurity requirements. The security design review should be comprehensive, covering all system components to detect weak connections between different subsystems.

Cybersecurity Assurance Planning ($P_4$)

Review Supplier Cybersecurity Assurance Plan ($P_4A_1$). It is essential to review the cybersecurity assurance plan of suppliers and confirm that it satisfies the requirements defined earlier. Suppliers should report the performance of any subcontracts related to the project. Exercising control over the supply chain by auditing the supplier’s security practices can help identify security loopholes early during the development stage.

Plan Cybersecurity Monitoring ($P_4A_2$). ISO/SAE 21434 and UNECE WP.29 advocate for cybersecurity monitoring during the operation of vehicles, which requires proper planning. Automotive organizations should identify and prepare the needed requirements that enable cybersecurity monitoring successfully. This includes identifying the type of information needed to monitor the state of the system. As attacks against automotive systems are evolving quickly, internal information (e.g., vulnerability assessment results) is not solely effective for cybersecurity monitoring. External information gathered by various industry players and government resources is needed to enhance the monitoring capabilities. Any information should be analyzed in the right context and using proper means to be helpful. Thus, before moving automotive
systems to operation, all the tools needed to analyze the data should be prepared. 

**Plan Cybersecurity Incident Response (P₄A₃).** The automotive industry should be prepared to respond promptly to exposed vulnerabilities and cyberattacks without jeopardizing vehicles’ safety. Automotive systems are prone to various cyberattacks, requiring a different course of action depending on their type and severity. Hence, the automotive industry should be ready to assess the risk level and have an appropriate plan according to the severity and attack type. Moreover, automotive organizations should plan for safe Over-the-Air (OTA) security patches needed to address exploited software vulnerabilities.

**Cybersecurity Implementation (P₅)**

**Follow Security Coding Practices (P₅A₁).** Automotive systems’ code complexity causes development challenges, leaving a high margin of vulnerabilities if not properly managed. Following well-established security coding practices supports the development processes by offering guidelines that help prevent common bad practices. Following ISO/SAE 21434 and UNECE WP.29 recommendations, the automotive industry should comply with MISRA C guidelines, specifically MISRA-C:2012 (Guidelines for the Use of the C Language Critical Systems), which defines 159 rules. These rules touch different topics, including functions and libraries prone to failure, naming conventions, and complexity limits. Another growing automotive coding standard is AUTOSAR which aims to standardize the software architecture of ECUs and improve program traceability. Developers should be trained ahead of time to avoid delivery delays.
Vulnerability and Threat Analysis ($P_3A_2$). By this phase, security engineers acquire enough knowledge about the underlying development details of the system. Hence, they can conduct a thorough vulnerability and threat analysis to identify security flaws in the design and implementation phases. Vulnerability analysis can be conducted based on historical vulnerabilities and by utilizing security vulnerability metrics that measure every component’s vulnerability score [20]. Moreover, threat analysis should examine potential attack paths at this phase, giving a more precise estimation of the risks.

Prioritize Security Testing ($P_3A_3$). Ensuring efficient and reliable security testing for automotive systems requires prioritizing components and interfaces based on their fragility and criticality. Vulnerability and threat analysis results can guide security testers toward the weakest component that requires intensive evaluation. Testing components that are more likely to contain defects first boosts vulnerability identification. Recommended by ISO/SAE 21434, this prioritization enables security engineers to quickly identify the largest number of defects and errors. Thus, it allows sufficient time to resolve the recognized issues. Chapter 4 of this thesis presents security vulnerability metrics. The proposed security metrics can quantitatively identify the weakest components of automotive systems, allowing security engineers to prioritize the testing towards the most vulnerable entities.

Cybersecurity Component Testing ($P_6$)

Static Code Analysis ($P_6A_1$). It is essential to review and analyze the code for defects that leave the system vulnerable. Manual code inspections are incompetent, considering the complexity and size of automotive systems. Static code analysis tools are
efficient tools that automatically and comprehensively analyze the source without re-
quiring security testers’ intervention. They can be executed during the development
of components, enabling prompt fault detection and mitigation. Static code analysis
tools examine the automotive source code against a set of coding rules to identify
common coding defects like buffer overflows and null pointer dereference. Moreover,
static code analysis tools run seamlessly and validate the code against MISRA and
AUTOSAR coding guidelines, assuring a compliant system(111,705),(991,764).

Software Composition Analysis (P6.A2). To validate the automotive system against
known vulnerabilities, security engineers can utilize software composition analysis
tools that automatically analyze the source code against a database of reported vul-
erabilities. Software composition analysis helps manage opensource code and en-
sures security by performing automated scans that identify opensource components
(including libraries) and validate their license and security vulnerability. Moreover,
these tools can analyze binaries enabling the validation of outsourced components
and legacy code against known vulnerabilities without causing any overhead.

Dynamic Analysis (P6.A3). While static code analysis is helpful to identify typical
defects, it cannot detect runtime vulnerabilities. Dynamic analysis tools examine au-
tomotive components by running and monitoring their execution to detect incorrect
behaviors. Dynamic analysis can identify critical defects that can obstruct the func-
tionality of vehicles, like memory leaks and concurrency errors. Moreover, it can help
identify weaknesses in outsourced code as it analyzes binary files.

Vulnerability Scanning (P6.A4). As attacks against automotive systems are becoming
more common, it is vital to ensure that the system is free of known vulnerabilities.
that helped establish attacks. Vulnerability scanning tools validate automotive systems against existing attack patterns and vulnerabilities. These tools are essential to avoid legacy vulnerabilities derived from old code and cause security decay, affecting automotive systems’ robustness against cyberattacks. Vulnerability scanning tools direct the testing to various automotive systems elements, including ECU software, configurations, ports, and interfaces.

Grey-box Fuzz Testing \((P_{6A_5})\). Designing comprehensive test cases capable of validating the automotive system against all possible inputs is challenging. Fuzz testing examines automotive systems against various malformed inputs to trigger unexpected behaviors. It is a fast testing tool that requires minimum intervention from security testers. Grey-box fuzz testing is efficient for cybersecurity component testing as it guarantees sufficient code coverage without increasing testing overhead. Grey-box fuzz testing tools acquire knowledge about the system by monitoring test case execution, enabling the generation of new inputs that can help maximize code coverage. Chapters 5 and 6 offer grey-box and hybrid fuzz testing frameworks that manage the testing challenges of automotive systems. The presented fuzz testing frameworks support security engineers in identifying security defects by exhaustively and efficiently validating the most vulnerable components of automotive systems.

Cybersecurity Integration Testing \((P_7)\)

Review Supplier Cybersecurity Compliance \((P_{7A_1})\). Before integrating outsourced components into the automotive system, it is essential to ensure that suppliers complied with the predefined security requirements. At this stage, security engineers should review supplier’s reports and security testing results. Exceptions and gaps
should be addressed according to the predefined policies.

**Black-box Fuzz Testing (P<sub>7</sub>A<sub>2</sub>).** One of the tools that attackers use to penetrate automotive systems is black-box fuzzers. Black-box fuzzy testing blindly validates the automotive systems against numerous generated test cases. Such a testing methodology does not cost any overhead, and it is efficient to validate outsourced components as it requires only a binary file. Black-box fuzz testing can avoid security decay caused by legacy code that is challenging to test. Without requiring any knowledge about the legacy components, black-box fuzzers traverse legacy code with numerous test cases that exploit faults that attackers attempt to find. Currently, several protocol-based fuzzers can be used for cybersecurity integration testing, including Controller Area Network (CAN), Wi-Fi, Unified Diagnostic Services (UDS), and Bluetooth.

**White-box Penetration Testing (P<sub>7</sub>A<sub>3</sub>).** One of the most potent testing methods for automotive systems is penetration testing. It assesses the system’s strength comprehensively, including its software components and network communication, making it an effective technique for cybersecurity integration testing. In white-box penetration testing, security testers are provided detailed information about the system, making the testing less time-consuming and more effective in defect identification.

**Cybersecurity Resilience Verification (P<sub>8</sub>)**

**Black-box Penetration Testing (P<sub>8</sub>A<sub>1</sub>).** At this step, security testers impersonate attackers and try to penetrate the system without acquiring any knowledge about the system. Black-box penetration testing mimics a real-world scenario and identifies the potential capabilities of attackers. It is a time-consuming task that requires unique expertise, but it is powerful to validate the resilience of automotive systems.
3.2. SECURE VEHICLE SOFTWARE ENGINEERING (SVSE)

Threat Analysis and Risk Assessment (P\textsubscript{8}A\textsubscript{2}). Final threat analysis and risk assessment are required before moving the vehicle to operate. The security testing results and implemented measures to prevent threats should be considered at this stage to estimate the security risk during vehicles lifespan.

Cybersecurity Maintenance (P\textsubscript{9})

Monitor and Respond to Cyberattacks (P\textsubscript{9}A\textsubscript{1}). During the development stage, the SVSE lifecycle paved the road for cyberattack monitoring. At this stage, automotive organizations are expected to employ the proper means to monitor and detect cyberattacks continuously. Attacks and threats should be traced to identify the affected code region and mitigate the vulnerabilities. Automotive organizations should follow the best practices to release OTA security patches that resolve identified vulnerabilities. Despite all the efforts to prevent cyberattacks, attackers might succeed in their mission. In this case, automotive organizations should put into action the response plans to assure prompt feedback that limits the damages. Records of successful attacks should be created and maintained to prevent their existence in future projects.

Identify and Mitigate Security Decay (P\textsubscript{9}A\textsubscript{2}). Automotive organizations should continuously evaluate vehicles’ security robustness to avoid security decay that exposes autonomous vehicles to malicious behavior. Both ISO/SAE 21434 and UNECE WP.29 principles warn automotive organizations of security decay’s criticality during vehicles’ long lifespan. During the operation phase, vulnerability measurement and threat assessment should be periodically applied to estimate the risks according to newly discovered vulnerabilities and evolved attackers’ capabilities. According to
the risk, proper actions should be applied to assure safe and reliable vehicles. The Autonomous Vehicle Security Decay Assessment (AVSDA) presented in Chapter 7 can aid security engineers in identifying security decay during the development stage and operation stages. AVSDA considers the evolving threats and the implemented security practices to determine a change in the resilience of automotive systems.

3.3 Summary

CAVs promise a future of safe driving, relying on numerous advancements that prevent accidents. Nevertheless, this innovation that aims to enhance transportation entails critical cybersecurity challenges that should be addressed during the development process. This chapter defines VSE, extensively compares and contrasts current widely used software engineering models (i.e., Verification and Validation Model (V-Model) and Agile model), and examines their strengths and weaknesses. Based on our analysis, we can conclude that the V-Model is more flexible in handling software integration, making it more suitable for this industry. This analysis helps guide software engineers to choose the model that can best match the requirements of this field.

Contrary to existing standards and research work that discuss security at the whole system level, this chapter also proposes a Secure Vehicle Software Engineering (SVSE) lifecycle devoted to the software level while considering the recommendations and requirements of vehicle standards. The SVSE incorporates security activities that address the automotive security challenges, ensuring a smooth development process. Moreover, the SVSE lifecycle addresses security irrespective of other requirements to
guarantee thorough considerations. However, it follows the traditional ASPICE V-Model, which is also compatible with ISO 26262 safety development lifecycle, enabling the three development cycles to run simultaneously. The SVSE development stage runs in parallel with the verification phases of the ASPIC V-Model. The former’s test stage executes with the validation phases of the latter, assuring security-by-design.

The remaining chapters of this thesis present practical security solutions that should be used during the SVSE lifecycle. Chapter 4 grants security vulnerability metrics that can be used during the development stage of the SVSE lifecycle. The metrics consider the unique architecture of CAVs while quantitively measuring the vulnerable software components. Chapters 5 and 6 present fuzz testing frameworks that can be employed during the testing stage of the SVSE lifecycle. Chapter 5 offers a vulnerability-oriented fuzz testing framework that prioritizes the grey-box fuzzing towards the weakest components of the automotive system, maximizing the vulnerability identification exposure process. Chapter 6 augments the grey-box fuzzing and explores the deep paths of the automotive systems using a hybrid fuzz testing approach that combines fuzzing and concolic exploration into one framework. Finally, Chapter 7 presents an Autonomous Vehicle Security Decay Assessment (AVSDA) framework that examines the system’s security risk over vehicles’ lifespan. AVSDA can be employed during the operation stage and cybersecurity decay review of the SVSE lifecycle.
Security Vulnerability Metrics for CAVs

Security testing is a vital phase in any Secure Software Development Lifecycle (SSDL) to diminish the vulnerabilities of automotive software and limit attackers’ chances. However, considering the size and complexity of the system, thorough validation of all the components of automotive systems is an infeasible task [28, 58]. As software components pose different security risks, ISO/SAE 21434 [4] requires the automotive industry to prioritize the testing.

Security metrics are commonly used to identify system vulnerabilities and guide the testing. Some researchers study the factors in a software system that can contribute to vulnerabilities. Medeiros et al. [141] investigate the effectiveness of using software metrics to identify vulnerabilities in a system. Their results confirm a strong relationship between software metrics and security vulnerabilities. Shin et al. [142] propose the use of complexity, code churn, and developer activity metrics as early signs of vulnerabilities. Similarly, Chowdhury et al. [70] assess the use of code complexity, code coupling, and cohesion metrics to recognize vulnerable files in a system.

Though these metrics are widely adopted and have successfully identified many vulnerable components, utilizing them in the automotive industry requires tailoring
4.1 SECURITY METRICS

In this chapter, we fill the gap by proposing security metrics specifically designed to handle the unique structure of CAVs. The security metrics aim to assist security engineers during the development life-cycle by identifying the weak links that keep the vehicle at high-security risks. We start this chapter by discussing the security vulnerability metrics and describing how each metric can be quantitatively measured. We then evaluate the security metrics on OpenPilot, an autonomous driving system. Finally, we compare the performance of our security metrics with other existing metrics.

4.1 Security Metrics

Security metrics are used as an indicator for vulnerabilities, threats, operational activities and to assist security experts in validating the protection mechanism in a system [143]. Based on our exhaustive study of the unique characteristics of automotive systems [12] and the various successful attacks conduct on vehicles [144–146], this section propose security vulnerability metrics tailored for CAVs.

The proposed metrics measure vulnerabilities at the Software Component (SWC) level. An SWC within autonomous systems represents a piece of code that implements an application or part of an application. Automotive-SPICE [35], a technical standard for autonomous software development, defines software components as “the lowest element level of the software architecture for which the detailed design is defined”. At the same time, AUTOSAR standard outlines different types of SWCs, i.e., Application Software Components, Composition Software Components, and Service Proxy Software Components. Since not all industry players adopt AUTOSAR, we
define SWC more generally as a structural element that provides an interface. It can utilize different automotive communication means and is connected to other parts to fulfill a function. This includes all types of SWCs defined by AUTomotive Open System ARchitecture (AUTOSAR) [8], covering all kinds of embedded hardware and firmware in a vehicle system.

The security metrics are (1) Code Complexity, (2) Component Coupling, (3) Input and Output Data Vulnerability, (4) Past Security Issues, and (5) Component Maturity. All the metrics can be evaluated simultaneously, and the results of these metrics are utilized in the final assessment, the security vulnerability score. We elaborate on the importance of each metric in what follows and describe how they can be calculated.

4.1.1 Code Complexity

Integrating millions of code lines in vehicles enabled them to become more aware of their environment. However, the code complexity of autonomous vehicle systems can increase the number of defects. Many researchers associated code complexity with the existence of vulnerabilities [70, 142, 147]. Complex code is challenging to understand, test, validate, and maintain. Attackers look for defects in the system that can be exploited. Hence, complex code increases attackers’ chances and is a good indicator of a high number of vulnerabilities. Researchers propose different attributes to calculate code complexity, including Source Line of Code (SLOC), Nesting Count, Nesting Depth (ND), McCabe’s Cyclomatic, and Number of Children (NOC) [148].

Durisic et al. [147] collaborated with Volvo Car Corporation to assess whether code complexity metrics can guide developers and security testers toward the most complex
SWCs of autonomous systems. Their study shows that code complexity and coupling can efficiently be used in the automotive industry. In general, developers consider Nesting Count, Nesting Depth (ND), and lack of structure to be the attributes that most reflect complexity in a system [149]. Moreover, ND and Number of Children (NOC) correlate to vulnerabilities the most [70, 142]. Accordingly, we define Code Complexity (CX) as a combination of Source Line of Code (SLOC), ND, and NOC. The CX of component $C$ can be calculated using Equation 4.1.

$$CX(C) = \omega_1 SLOC + \omega_2 ND + \omega_3 NOC$$ (4.1)

### 4.1.2 Component Coupling

Coupling between objects is the concept that two or more entities rely on each other to fulfill functionality. In the automotive industry, code coupling can support engineers in finding complex components that might require more attention and testing than other parts in the system [147]. Code coupling is not only used to recognize complexity but is also extensively used to identify vulnerabilities [70, 141, 150, 151]. The dependability of entities in a system can help a malicious message propagate from one component to another, making an attack impact more severe.

In autonomous vehicle systems, coupling can be at two levels: components and

---

1Weights are used to assign the degree of importance to the attributes. Values assigned to the weights should fall within a predefined range and be used consistently for the same degree of importance. For example, the highest importance weight can have a value of 8, medium importance can have a value of 4, and low importance can have a value of 1.
functions. We covered function coupling in the code complexity. Component Coupling (CC) aims to measure how reliant a component is on other subsystems. Communication between autonomous systems components is needed to offer customers various functionalities. For example, the safety system in modern vehicles can communicate with the central locking system to lock the doors when the vehicle reaches a certain speed and unlock them when it stops. Such communication between components is essential to ensure the safety of passengers. However, components coupling permits the propagation of malicious messages [152].

We define CC as the number of Electronic Control Units (ECUs) reachable from a component’s ECU, calculated using transitive closure. The transitive closure determines direct and indirect coupling. For example, consider component A which runs on $ECU_A$ and connects through the gateway ECU to components $B$ and $C$ through $ECU_B$ and $ECU_C$, respectively. Component $A$ does not rely on component $D$, so no communication between $ECU_A$ and $ECU_D$ occurs. However, component $B$ communicates with component $D$. Consequently, a malicious message can propagate indirectly from $ECU_A$ to $ECU_D$ through $ECU_B$. The CC of component $C$ is calculated using Equation 4.2, where $R$ is the number of relationships between the ECUs of Component $C$.

$$CC(C) = \bigcup_{i=1}^{\infty} R^i$$  \hspace{1cm} (4.2)

4.1.3 Input and Output Data Vulnerability

Components of autonomous systems operate based on the collected data from sensors, radars, cameras, vehicles, infrastructure, users’ mobile devices, and other
sources. According to the inputs received and the embedded functionality, a component will transmit signals that control the vehicle’s behavior. Inputs and outputs offer an exceptional opportunity for attackers. Vehicles’ diverse operating conditions make data validation a challenging job. Vehicles are always moving and sensing their surrounding environment. Hence, it is impossible to quantify all possible inputs and outputs that an autonomous system can receive and send.

Inputs and outputs are transmitted through different communication means. For example, autonomous systems depend on data received by the Global Positioning System (GPS) receiver to identify a vehicle position and navigate drivers to their destinations. Vehicle to Vehicle (V2V) communication is used to distribute traffic information. GPS and V2V channels each pose different risks on CAVs. GPS is vulnerable to jamming and spoofing [153], while V2V communication exposes the vehicle to external attacks like eavesdropping, spoofing, Denial of Service (DoS), and spamming [154]. While both of these communication means put the vehicle at risk, the level of threat between one communication means and the other is different. V2V communication exposes the autonomous system to a broader range of attacks [154].

The Input and Output Data Vulnerability (DV) observes two elements: the type and the mean of communication used. Fixed inputs and outputs data types (e.g., data types with constant values) are easier to validate and considered less risky compared to fluctuating inputs and outputs data types (e.g., an integer value that has an extensive range) that are challenging to validate. Moreover, different communication technologies are subject to various security issues. Thus, each communication mean is assigned a weight\(^1\) according to its criticality. The DV of component \(C\) can be calculated using Equation 4.3. \(K\) represents the total number of communication
4.1. SECURITY METRICS

means, $FI$ and $FO$ represent fixed inputs and outputs, respectively, $LI$ and $LO$ represent fluctuating inputs and outputs respectively, $\omega_k$ is the weight of a specific communication mean, and $\chi$ and $\upsilon$ are weights of fluctuating inputs and outputs.

$$DV(C) = \sum_{k=1}^{K} \omega_k |FI(C)| + \chi \omega_k |LI(C)| + \omega_k |FO(C)| + \upsilon \omega_k |LO(C)|$$

(4.3)

4.1.4 Past Security Issues

There have been many successful attacks against CAVs [3]. News of a security breach often gets the attention of malicious users who take advantage of an exposed vulnerability to conduct similar events. Hence, any bug, vulnerability, or attack on the vehicle software system must be carefully examined to prevent future malicious actions. Past Security Issue (PSI) gives higher importance to components that were subject to attacks.

PSI examines the frequency and age of an incident. A security incident that occurs regularly indicates a weakness in the system. Thus, attacks that happen many times are given higher importance. Attacks that arose from a long time ago and did not recur are more likely resolved. Hence, PSI introduces the forgetting factor to give more importance to recently discovered vulnerabilities. Equation 4.4 illustrates how the PSI of component $C$ can be calculated. $Y$ represents the total number of years since the first vehicle attack, $A_y$ represents the number of attacks that occurred in year $y$, and $\lambda$ is the forgetting factor.

$$PSI(C) = \sum_{y=1}^{Y} A_y \lambda^{Y-y} \mid 0 \leq \lambda \leq 1$$

(4.4)
4.1. SECURITY METRICS

4.1.5 Component Maturity

Component Maturity (CM) is essential for identifying vulnerabilities and security decay during vehicle operation. A component can witness many changes due to requirements changes, enhancements, security updates, and bug fixes. Researchers observe that continuous updates and code changes can weaken code robustness and make it more prone to vulnerabilities [142, 155]. Code Churn (CCH) calculates the modifications made to a component over time and quantifies the changes’ extent. We evaluate CCH by identifying the ratio of changes in a component, including deleted, added, and modified SLOC, as presented in Equation 4.5.

As we are interested in evaluating a component’s security decay, it is vital to exclude the changes that are meant to enhance the security of an element while calculating CCH. Reviewing the security practices developed within a component can enhance the security measures and improve the component’s defense mechanism. We consider components that witness security improvements as more resilient against cyberattacks. We calculate the Security and Maintenance Intensity (SMI) by counting the security enhancement activities since a product release. The reverse percentage is used to determine a low risk for proper security-maintained components, as shown in Equation 4.6. Therefore, CM covers extensively changed and low security-maintained code. The CM of component $C$ can be calculated following Equation 4.7.

$$CCH(C) = \frac{|\text{Changed SLOC}|}{|SLOC|}$$  \hspace{1cm} (4.5)
4.2 EVALUATION

\[ SMI(C) = 1 - \left( \frac{\text{Security Maintenance Activity}}{\text{Age}} \right) \]  (4.6)

\[ CM(C) = CCH(C) + SMI(C) \]  (4.7)

4.1.6 Security Vulnerability Score

The final assessment of a component’s Security Vulnerability (SV) is calculated based on the values obtained from the previous five steps. As presented in Equation 4.8, to have proportional values, the results obtained from each metric for a component \( C \) are divided by the maximum (\( MAX \)) value that can be acquired by the corresponding metric covering all components of the system. Different weights \( \alpha, \beta, \gamma, \delta, \theta \) can be assigned to each security metric.

\[
SV(C) = \alpha \left( \frac{CX(C)}{MAX(CX)} \right) + \beta \left( \frac{CC(C)}{MAX(CC)} \right) + \gamma \left( \frac{PSI(C)}{MAX(PSI)} \right) \\
+ \delta \left( \frac{DV(C)}{MAX(DV)} \right) + \theta \left( \frac{CM(C)}{MAX(CM)} \right) \]  (4.8)

4.2 Evaluation

This section demonstrates the use of security metrics. We utilize OpenPilot (Version 0.7.8) [156], in our evaluation. OpenPilot is an open-source, driving, and safety
assistant system developed by comma.ai [157]. It offers SAE Level 2 driving assistance capabilities fulfilling the functions of Adaptive Cruise Control (ACC), Automated Lane Centering (ALC), Forward Collision Warning (FCW), and Lane Departure Warning (LDW). It supports various vehicle models, including Honda, Toyota, Hyundai, and Lexus. The automotive system also offers safety features by implementing Driver Monitoring (DM) functionality that warns inattentive drivers.

OpenPilot has one component only, Autopilot. We apply the metrics on the Autopilot component, illustrating their usefulness. Then we quantitatively evaluate the proposed security metrics’ effectiveness in identifying vulnerabilities in OpenPilot and compare them with other existing metrics [70,142].

4.2.1 Applying the Security Metric to OpenPilot

This assessment verifies the applicability of the designed metrics. We demonstrate how the security metrics can be utilized in autonomous systems.

**Code Complexity.** Following Equation 4.1, the Source Line of Code (SLOC), Nesting Depth (ND), and the Number of Children (NOC) are used to measure the Code Complexity (CX) metric of OpenPilot. The component has a total of 52,608 SLOC, 6,298 ND, and 6,148 NOC. Since ND and NOC are associated with vulnerabilities, we give them weights of 2 and 3, respectively [70,142]. Equation 4.9 shows the calculation of the CXR metric.

\[
CX(OpenPilot) = 52,608 + 2(6,298) + 3(6,148) = 83,648 \quad (4.9)
\]
Table 4.1: OpenPilot inputs and outputs

<table>
<thead>
<tr>
<th>Data</th>
<th>Communication Mean</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Serial</td>
<td>242</td>
</tr>
<tr>
<td>Input</td>
<td>CAN</td>
<td>407</td>
</tr>
<tr>
<td>Input</td>
<td>GPS</td>
<td>397</td>
</tr>
<tr>
<td>Input</td>
<td>V2I</td>
<td>1</td>
</tr>
<tr>
<td>Input</td>
<td>U2V</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>Serial</td>
<td>15</td>
</tr>
<tr>
<td>Output</td>
<td>CAN</td>
<td>73</td>
</tr>
<tr>
<td>Output</td>
<td>V2I</td>
<td>211</td>
</tr>
<tr>
<td>Output</td>
<td>U2V</td>
<td>1</td>
</tr>
</tbody>
</table>

**Component Coupling.** To evaluate the component coupling (CC) metric, we reviewed the CAN messages sent from OpenPilot. Our analysis shows that the component communicates with the Engine Control Module, Brake Control Module, Safety System, Seat Control Unit, Powertrain Control Module, Transmission Control, Telematics Control Unit, Active Front Steering, and Battery Junction Box. Between all these components, the Autopilot system is highly dependent on the Engine Control Module and Brake Control Module. Equation 4.10 presents the CC metric calculation, and the ECUs have direct coupling only.

\[ CC(OpenPilot) = 9 \] (4.10)

**Input and Output Data Vulnerability.** OpenPilot inputs and outputs are all fluctuating, and the main mean of communication is the Controller Area Network (CAN). The automotive system receives and sends data using serial communication, CAN, Vehicle to Infrastructure (V2I), and User to Vehicle (U2V). Each of these communication means poses different risks. Accordingly, we assign different weights for these communication means, as shown in Equation 4.11. Table 4.1 summarizes
Table 4.2: OpenPilot bugs by years

<table>
<thead>
<tr>
<th>Year</th>
<th>Number of Issues</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018</td>
<td>1</td>
<td>Resolved</td>
</tr>
<tr>
<td>2019</td>
<td>17</td>
<td>Resolved</td>
</tr>
<tr>
<td>2020</td>
<td>42</td>
<td>Resolved</td>
</tr>
<tr>
<td>2019</td>
<td>1</td>
<td>Unresolved</td>
</tr>
<tr>
<td>2020</td>
<td>8</td>
<td>Unresolved</td>
</tr>
</tbody>
</table>

the number of received and sent messages of each channel.

\[
DV(OpenPilot) = 242 + 2(407) + 3(397) + 5(1) + 5(1) + 15 + 2(73) + 5(211) + 5(1) = 3,478
\] (4.11)

**Past Security Issues.** This metric measures the number of vulnerabilities in the Autopilot component, giving higher importance to recently discovered ones. OpenPilot has been in production since 2016, but the bugs documentation on GitHub started in December 2018. Table 4.2, summarizes the reported bugs. In total, there are 69 reported bugs in the system. Equation 4.12 represents the Past Security Issues (PSI) metric calculation, which gives more importance to attacks that occurred in 2020.

\[
PSI(OpenPilot) = (0.5)^2 + 18(0.5^1) + 50(0.5^0) = 59.25
\] (4.12)

**Component Maturity.** The percentage of changes in a component is identified to measure the Component Maturity (CM) metric. CM also examines low security-maintained code. Within OpenPilot, 30,688 SLOC is modified, representing around 58% of the code for four years of operation. Moreover, none of the applied changes
are labeled as security enhancement or maintenance. Equation 4.13 depicts the calculation of the CM metric.

\[
CM(\text{OpenPilot}) = 100 \left( \frac{30,688}{52,608} \right) = 58
\]  

(4.13)

### 4.2.2 Comparative Analysis of Vehicle Metrics and Other Metrics

This section verifies the proposed metrics’ effectiveness by comparing the files’ vulnerability scores with the number of discovered vulnerabilities in every file. We calculate the metrics of every file in OpenPilot. In total, as of October 2020, OpenPilot has 425 files and 60 documented resolved bugs. We reviewed the reported bugs and linked 24 bugs to the system files. We compare the performance of the used metrics with two other sets of metrics. One set is code complexity and churn metrics [142], and the other set is code complexity, code coupling, and cohesion metrics [70]. We identify true-negative, true-positive, false-positive, and false-negative cases. Then, we measure accuracy, precision, and recall rates defined as follows:

- **True-negative (TN)** is the count of files identified as not vulnerable, with no reported vulnerabilities.
- **True-positives (TP)** is the count of files identified as vulnerable, with reported vulnerabilities.
- **False-negative (FN)** is the count of files identified as not vulnerable, while some reported vulnerabilities exist.
- **False-positive (FP)** is the count of files identified as vulnerable, while no reported vulnerabilities exist.
4.2. EVALUATION

- Accuracy shows the overall correct identification rate, evaluated using Equation 4.14.

\[
Accuracy = \frac{TN + TP}{TN + TP + FN + FP}
\]  
(4.14)

- Precision shows the effectiveness of vulnerability identification, evaluated using Equation 4.15.

\[
Precision = \frac{TP}{TP + FP}
\]  
(4.15)

- Recall shows the detection rate of vulnerable components, evaluated using Equation 4.16.

\[
Recall = \frac{TP}{TP + FN}
\]  
(4.16)

The higher the values of these rates, the better performant are the metrics.

The results are summarized in Table 4.3. The proposed metrics outperform the other approaches in accuracy, precision, and recall. Our metrics achieve a 78% recall ratio indicating that they can identify vulnerable files efficiently. The vehicle metrics had a notably high ratio of 94% for accuracy, indicating that the overall vulnerability identification is correct. Though the precision ratio of the proposed metrics is better than the other approaches, it is relatively low. This means that the number of files recognized as vulnerable and do not possess any vulnerability is high. While this causes extra unneeded work, having more false-positive cases to enhance the true-positive results is better in vulnerability identification.

We further analyze the performance of the proposed metrics by examining the relationship between the average metrics ratio and the number of bugs reported in a file as shown in Fig. 4.1. The highest number of reported bugs in a file is 6, which vehicle metrics identifies as the most vulnerable with a ratio of 1. As shown in Fig.
4.3. SUMMARY

Table 4.3: Comparison between vehicle metrics and other metrics

<table>
<thead>
<tr>
<th></th>
<th>Vehicle Metrics</th>
<th>Complexity, and Code Churn Metrics [142]</th>
<th>Complexity, Coupling, and Cohesion Metrics [70]</th>
</tr>
</thead>
<tbody>
<tr>
<td>True-negative</td>
<td>390</td>
<td>395</td>
<td>344</td>
</tr>
<tr>
<td>True-positive</td>
<td>11</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>False-negative</td>
<td>3</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>False-positive</td>
<td>21</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>Accuracy</td>
<td>94%</td>
<td>94%</td>
<td>83%</td>
</tr>
<tr>
<td>Precision</td>
<td>34%</td>
<td>30%</td>
<td>12%</td>
</tr>
<tr>
<td>Recall</td>
<td>78%</td>
<td>50%</td>
<td>71%</td>
</tr>
</tbody>
</table>

Figure 4.1: Relationship between average metrics ratio and number of vulnerabilities

4.1, the Security Vulnerability (SV) assigned by the vehicle metrics is proportional to the files’ number of vulnerabilities. The higher the number of vulnerabilities, the higher is the SV value. In contrast, the other two sets of metrics \[70, 142\] show more arbitrary behavior where files with three reported bugs are assigned a higher vulnerability value than files with six bugs.

4.3 Summary

This chapter introduces security vulnerability metrics designed to identify the weak components of automotive systems that leave CAVs vulnerable to attacks. Early
identification of vulnerabilities can aid security engineers in prioritizing the security testing and identifying the security loopholes of the system before a vehicle production. The five metrics are (1) Code Complexity, (2) Component Coupling, (3) Input and Output Data Vulnerability, (4) Past Security Issues, and (5) Component Maturity.

We evaluated the metrics’ performance by experimenting with their usefulness in identifying vulnerabilities of OpenPilot, an Autopilot system. The results show that the metrics can identify vulnerabilities with an accuracy rate of 94%. The accuracy of our experiment may be affected by the number of identified vulnerabilities in OpenPilot. For example, not all the reported vulnerabilities of OpenPilot were linkable to a file. Moreover, since OpenPilot is relatively new (2018), the existing vulnerabilities might not be expansive and reflect the actual number of existing vulnerabilities. Yet, the proposed metrics outperform two existing sets of metrics in identifying automotive vulnerabilities with better accuracy, precision, and recall rates. Moreover, the proposed vehicle metrics successfully assign the highest security vulnerability value to the file with the most reported vulnerabilities. This evaluation shows the usefulness of the security vulnerability metrics in systematically and efficiently identifying the weak components of the system.

The subsequent chapters demonstrate further the usefulness of the proposed security metrics. Chapters 5 and 6 incorporate the metrics into security testing frameworks to guide the testing towards the weakest components. Chapter 7 employs the security vulnerability metrics in a risk assessment framework to identify vulnerable components.
Chapter 5

Grey-box Fuzz Testing for Vehicle Systems

Before a vehicle is released, security engineers need to verify the system’s security to avoid catastrophic incidents. Software security testing is a scalable and practical approach to identify systems’ weaknesses and vulnerabilities at an early stage and throughout their life-cycle. Recommended by the Road Vehicles Cybersecurity Engineering standard ISO/SAE 21434 [4], fuzz testing is one of the most efficient techniques to identify vulnerabilities quickly and without causing overhead. Though there are three different types of fuzz testing: black-box, grey-box, and white-box, the automotive industry currently adopts mainly black-box fuzz testing only [15].

Some researchers [14, 63, 158–160] utilize fuzz testing to discover vulnerabilities within vehicle software systems. However, only a few works introduce fuzz testing tools explicitly designed for the vehicle industry [14, 63, 159, 161]. Research efforts in this area are limited to evaluating and studying the applicability of black-box fuzz testing for CAVs [58, 95–97]. Black-box fuzzers are efficient considering the size and complexity of automotive systems. Nevertheless, adopting such a testing methodology for an automotive system is not a reliable solution. Black-box random fuzzing cannot provide a complete picture of which components are tested. The lack
of guidance in black-box testing can waste hours of testing on a single component, hindering a thorough evaluation of a safety-critical system. For this reason, the vehicle industry needs a software security testing solution that can facilitate the testing process, simulate the environment of vehicles, and target vulnerabilities.

Our research aims to fulfill these requirements by refining security assurance in the vehicle industry and easing security engineers’ jobs. This objective is achieved by designing an innovative grey-box fuzz testing framework that optimizes the vulnerability exposure process while addressing security testing challenges in the vehicle industry. Grey-box fuzz testing is a robust security mechanism that accumulates information about the system without increasing testing complexity, enabling fast and efficient security testing. The grey-box approach addresses three testing challenges: the system’s complexity and size by avoiding intensive code analysis, outsourcing by limiting the knowledge about the system, and input and output fluctuation by creating a massive number of inputs.

This chapter presents a vulnerability-oriented fuzz testing framework (VulFuzz) that prioritizes the testing toward weaker components of automotive software systems. First, we introduce VulFuzz engines that cooperatively work on achieving VulFuzz goals. Then, we evaluate the framework’s effectiveness in identifying unexpected behavior by applying it to an automotive subsystem.

5.1 Vehicle Grey-box Testing Framework

Vehicle software systems are complex software systems that rely on numerous technologies to operate and offer intelligent functionalities. Grey-box fuzz testing can evaluate a software component’s security using an extensive set of input combinations.
5.1. VEHICLE GREY-BOX TESTING FRAMEWORK

(a) Framework steps

(b) Framework engines (the steps automated by each engine are listed between parenthesis)

Figure 5.1: Vulnerability-oriented fuzz testing framework
We propose a vulnerability-oriented fuzz testing framework (VulFuzz) that validates a vehicle software system’s security with numerous valid inputs that strive to examine its vulnerable components thoroughly. The framework guides the testing toward the system’s most vulnerable or weak components by leveraging security vulnerability metrics that target vehicle software systems’ unique challenges. VulFuzz employs the vulnerability metrics to automatically identify the system’s weak or vulnerable functions\textsuperscript{1} and assigns a weight ($w$) based on the metric value. The higher the vulnerability score, the more security fragile the function is, and hence the higher the value of $w$. The framework gives high priority to weak functions and intensively examines them. Unlike other grey-box techniques, VulFuzz cares not only about coverage but also about the number of times a weak function is traversed. The weight assigned to functions identifies the threshold of testing. The framework is given a sample of good inputs to generate a range of valid test cases. VulFuzz runs each test case to monitor if it traverses a weighted function or has a connection to one. Such test cases permit the validation of vulnerable functions, so they are transferred to a high priority queue to create more test cases. In contrast, less attention is given to test cases that do not cover weak functions.

The framework is presented in Fig. 5.1. Fig. 5.1(a) shows the framework steps, which are automated by four engines presented in Fig. 5.1(b): Vulnerability Engine, Mutation Engine, Evaluation Engine, and Prioritization Engine. Each engine plays a vital role in accomplishing the steps of VulFuzz. The vulnerability engine measures functions’ vulnerability value. The mutation engine generates a range of valid inputs to examine the system. The evaluation engine assesses the usefulness of test cases. Finally, the prioritization engine prioritizes the testing toward vulnerable components.

\textsuperscript{1}Each function belongs to a particular component.
This section begins with an overview of the framework steps and then delves deeper into the engines of VulFuzz.

5.1.1 Framework Steps

The framework steps are presented in Fig. 5.1(a) and detailed as follows. The preparation for the fuzzing routine (steps 1, 2, and 3) runs at compilation time to minimize the overhead during the security testing phase. 1 First, the framework starts by calculating the security vulnerability value of each function using the source code of the system and assigns weights to vulnerable functions. 2 Then, the call graph of the system is generated. 3 Using sample inputs, the framework builds a dictionary to identify the input format. These samples are moved to the high priority queue to launch the testing.

The rest of the steps (4 to 9) can be combined in the fuzzing routine, as depicted in Algorithm 1. The routine is initiated during the security testing phase. 4 The framework begins by selecting a seed input from the high priority queue. If the high priority queue is empty, then the low priority queue is activated. If both queues are empty, the process terminates. 5 Next, the selected seed is mutated, and the program executes with the new input. 6 The framework updates the coverage table and the call count of weighted functions based on the seed execution. According to the results, the framework prioritizes the testing. 7 It adds the mutated input to the high priority queue if the test case traverses or has a path to a vulnerable function with a call count less than the assigned weight. 8 The vulnerability-oriented fuzz testing framework adds the mutated input to the low priority queue if it does not satisfy the high priority queue requirements but discovers new branches. 9 If the
conditions of both queues are not satisfied, the input is discarded. The fuzzing routine stops if VulFuzz cannot find more inputs that evaluate the vulnerable functions or expand the coverage. It also terminates if VulFuzz examines the vulnerable functions according to the assigned weights and reaches maximum coverage.

### Algorithm 1 Fuzzing Routine

```plaintext
while HighPriority ≠ ϕ OR LowPriority ≠ ϕ do
    if HighPriority ≠ ϕ then
        seed ← ChooseNext(HighPriority)
    else
        seed ← ChooseNext(LowPriority)
    end
    seed′ ← Mutate(seed)
    if seed′ IsVulnerableInteresting then
        add seed′ to HighPriority
    else if seed′ DiscoversNewBranch then
        add seed′ to LowPriority
    else
        seed′ ← ϕ
    end
end
```

5.1.2 Framework Engines

This subsection discusses the grey-box fuzz testing framework engines of Fig. 5.1(b) in detail, highlighting each engine’s role in the security testing framework.

**Vulnerability Engine**

The vulnerability engine is the core enabler of test guiding toward weak functions that require thorough evaluation. This engine is responsible for identifying the system’s components’ likelihood to have vulnerabilities and building the call graph as described in the following paragraphs.
Vulnerability Identification. The vulnerability engine, which runs during compilation time, takes the source code of an automotive system as an input and automatically measures the vulnerability scores to identify the functions that pose a high risk. The vulnerability score of each component is calculated based on the security metrics defined in Chapter 4 which are explicitly designed to identify vehicle software systems’ vulnerabilities.

\[
\text{result} = 0 \\
\text{if } x \geq 0: \\
\quad \text{result} = 100/x 
\]

Listing 5.1: An example showing the importance of weights

It is vital to test high-risk entities thoroughly to expose the system’s faults at an early stage. Existing grey-box testing techniques strive solely to expand code coverage without differentiating weak system entities. However, it is essential to examine certain functions many times. For example, consider the script presented in Listing 5.1; if \( x \) is assigned a value greater than 0, this script operates normally. However, when \( x \) holds a value of 0, this script raises an exception. Hence, coverage is not sufficient enough to expose some bugs in the system. Simultaneously, it is infeasible to test all the system’s entities several times within a specific time frame.

The security metrics guide us toward the complex components that require special treatment and intensive testing to maximize bug disclosure at an early stage. The higher the value of the overall security vulnerability metric, the more risk it likely poses. According to the security vulnerability of a function, a weight \( w \) is assigned that represents the number of times a function should be tested. Non-vulnerable functions are assigned a zero weight \( (w = 0) \).

Call Graph Building. The vulnerability engine creates the call graph at compilation
time since it is needed by the evaluation engine to direct the testing toward the
vulnerable functions. The call graph (CG) of a component \( C \) has a set of Nodes
\( N \) representing the total number of nodes in CG. Each node in CG represents a
function and a directed edge between two nodes \( (n \rightarrow n') \) demonstrates the possibility
of traversing from function \( n \) to function \( n' \).

**Mutation Engine**

The mutation engine’s goal is to generate seed inputs that pass the validation
criteria of automotive components to expand code coverage. Automotive compo-
nents communicate via the automotive Ethernet, Controller Area Network (CAN),
or Flexray buses. Entirely random mutation of the communication messages often
fails at the data validation step, leaving the code’s crucial parts without any valida-
tion. The mutation engine of the state-of-the-art fuzzer American Fuzzy Lop (AFL)
performs a small bit-level mutation on good inputs to generate a range of seed in-
puts. It is designed for compact data formats, e.g., multimedia files, images, and
compressed data [107]. Bit-level mutation presents some critical limitations when ap-
plied to systems that are format-specific like vehicle software systems [108]. Though
a bit-level mutation introduces a minor change that barely affects the input, the mu-
tation can ruin the input structure. Moreover, bit-level mutation fails to preserve
input data types. To overcome these challenges, VulFuzz mutation engine adopts an
input structure-aware mutation approach composed of three major parts: (1) input
format, (2) data type-based mutation, (3) crossover-based mutation. Before starting
the fuzzing routine, the input format is identified. Then VulFuzz passes seed inputs
to the mutation engine to perform data type-based mutation. After finalizing the
fuzzing process with the data type-based mutation, the mutation engine switches to crossover-based mutation to find good test cases and expand code coverage.

**Input Format.** Some solutions are proposed to reduce dropped messages and make the mutation structure-aware, including taint-based mutation, input parsers, and dictionaries [162]. Taint-based fuzzers require extensive code analysis that increases the testing overhead [163]. Input parsers adopted by grey-box fuzzers are used to identify input structures, guiding the mutation toward data chunks and preserving essential file headers. However, these input parsers work best on media files, video files, and web files [108]. In the vulnerability-oriented mutation engine, we utilize a dictionary for preserving the input format. Dictionaries are a robust technique broadly used to feed the fuzzer information about the inputs, improving fuzzing efficiency [107, 164]. The vulnerability-oriented dictionary marks the file header and fields essential to prevent inputs from dropping.

**Data type-based Mutation.** After identifying the input format, the mutation engine attempts to identify the data field types automatically. This step is required to perform data type-based mutations, which helps the seed inputs pass the initial validation steps and explore the system. Such a mutation technique triggers more bugs than random mutation as it smartly preserves the structure of messages, and at the same time, validates the system with a different input range [165].

For each seed input, the mutation engine performs one mutation operation on one field. We perform small mutations as we need to keep the majority content of seeds that helped explore the system and test vulnerable components. The mutation engine first tries to parse the field that needs mutation to a data type, e.g., numeric,
boolean, and string. According to the data type, a set of operations can be performed. For numerical data, the mutation engine is designed to randomly choose one of the following mathematical operations: subtraction, multiplication, division, and addition. For a given numerical field $F$, an arbitrary numerical field $A$ is generated to randomly apply one of the mathematical operations. The mutation engine mutates boolean data to either true or false. As for strings, we perform single-bit random deletion, insertion, or flipping. If the mutation engine fails to identify the data field type, it performs the classical random one-bit mutation [107]. Moreover, to test the system’s input validation routine, the mutation engine mutates fields with different data types (e.g., a numerical field is mutated to string). However, such validation is only performed once for each field to avoid halting the validation process and to explore the system.

_Crossover-based Mutation._ Motivated by genetic algorithms, several grey-box fuzzers use this type of mutation [107,108,166]. We statically swap chunks of different seeds to preserve the input structure. Given a seed $s$, we randomly choose a portion $p$, where $p_1$ and $p_2$ are the start and end indexes of this portion. Using the same indexes, another portion $p'$ is sliced from a random seed $s'$. Portion $p$ is then placed in the position of $p'$ in $s'$ and $p'$ is placed in the position of $p$ in $s$, generating two new seeds. We explicitly preserve the location of the swapped portion to maintain the format of seeds.

**Evaluation Engine**

_VulFuzz is guided toward vulnerable components and coverage expansion. The evaluation engine helps in achieving this objective by monitoring the performance
Figure 5.2: An example call graph showing a path to a vulnerable function of seed inputs. Inspired by AFL, for each seed input, the evaluation engine records the traversed edges. It utilizes lightweight instrumentation to detect branch coverage. Branch coverage offers substantially more insight into the execution path than statement coverage. It can identify the branches of conditional statements that cannot be recognized with simple statement coverage [167]. Coverage helps the fuzzer understand the fuzzing progress and identify the usefulness of a seed input.

To successfully direct the fuzzer toward vulnerable components, the evaluation engine detects the seed inputs that traverse or have a path to a vulnerable function. Using the weighted function created by the vulnerability engine, the evaluation engine identifies the vulnerable functions and monitors the test cases that traverse them. VulFuzz gives high importance to vulnerable functions and strives to validate their security thoroughly. Hence, even if a seed input does not traverse a vulnerable function, the evaluation engine examines whether this seed input can eventually reach vulnerable components. Inputs that traverse nodes connected to the vulnerable functions have a chance to reach them with a slight mutation. The call graph generated by the vulnerability engine is used to determine whether an executed input has a path that can reach a vulnerable function, excluding the system entry point. Given the call graph of Fig. 5.2, which has one vulnerable function $n_7$, a seed input has a
path to $n_7$ only if it traverses nodes $n_3$ or $n_6$. For example, consider a seed input $s_1$ that crosses nodes $n_1$, $n_2$, and $n_4$. Seed $s_1$ is unlikely capable of reaching node $n_7$. Consequently, it is marked as unbenevolent for testing vulnerable functions.

**Prioritization Engine**

In complex and large systems like vehicle software systems, test case prioritization is vital during the testing and validation phase. The vulnerabilities of the system are increasing with a limited time budget. Existing grey-box fuzzing techniques do not differentiate between test cases; they all reside in the same queue, executed in a first-come-first-served (FIFO) order. On the contrary, VulFuzz prioritizes test cases based on their discoveries; seeds that trigger vulnerable functions are given higher priority. The engine analyzes the coverage table and weighted functions count generated by the evaluation engine to determine whether a seed input should be added to the high priority queue, low priority queue, or disregarded. More than two queues can be utilized to target functions at multiple thresholds.

As discussed in the vulnerability engine, each identified vulnerable function is assigned a weight ($w$) to test weak functions thoroughly. Test cases that explore or have a path to vulnerable components and whose count is less than the assigned weight are highly necessary and thus added to the high priority queue. Test cases that do not execute or have a path to a vulnerable function but expand code coverage (discover at least one new branch that was not discovered earlier) are considered a low priority and are moved to the low priority queue. Lastly, test cases that do not explore new branches and do not execute vulnerable functions are not added to any queue.
Seed inputs that join a queue are assigned energy values to be further mutated and used as new inputs in the fuzzing routine. An energy value represents the number of times a seed input should be mutated. The prioritization engine adopts a constant energy assignment while giving more energy to seeds that explore vulnerable functions. High priority seed inputs that traverse vulnerable functions are given triple the energy of low priority seed inputs, allowing them to generate more inputs to provide a better chance for exploring vulnerable functions. Seeds that belong to the high priority queue, but do not traverse a vulnerable function, are assigned double the energy of low priority seeds. Such test cases have a high chance to traverse vulnerable functions, but they may never be able to reach them.

For example, consider Fig. 5.2, with a vulnerable node $n_7$ and whose execution count is less than its weight. A seed $s$ that traverses $n_7$ is assigned an energy value of $3x$, where $x$ is a constant defined by the security engineer. A seed $s'$ that executes nodes $n_1$ and node $n_3$ is given an energy value of $2x$. Hence, to save the fuzzing power on weak functions, test cases similar to $s'$ are assigned less energy value than those similar to $s$ that guarantee vulnerability exploration. On the other hand, test cases that belong to low priority queues are allocated lower energy values. A seed $s''$ that discovers edge $n_1 \rightarrow n_2$ for the first time, is assigned an energy value of $x$.

5.2 Implementation and Evaluation

To evaluate the effectiveness and performance of VulFuzz, we implement the framework and apply it to the automotive system OpenPilot. This section describes the evaluation setup, analyzes the obtained results, and compares them with two other fuzzing methodologies: AFL and Mutation-based fuzzer. This section also includes
an ablation study on the four engines of VulFuzz.

5.2.1 Evaluation Setup

We implement the framework and evaluate its performance on OpenPilot (Version 0.7.8) [156]. As a safety-critical system, OpenPilot requires intensive security testing to validate and verify the system’s solidity against malicious behavior. Fuzz testing generates an array of unexpected inputs that can trigger improper behaviors in the system. OpenPilot supports a regression testing tool, Process Replay [168], that executes the system processes and validates the output against a predefined input. To run the fuzz testing, we adjusted the tool to accept any input without limitations allowing a robust validation of the system’s security against unexpected inputs.

To verify the effectiveness of the vulnerability-oriented fuzz testing framework (VulFuzz), we compare our results to the state-of-the-art fuzzer American Fuzzy Lop (AFL) [107] and an unguided mutation fuzzer. OpenPilot is designed using both Python and C languages. The original AFL does not support Python language, so we used the Python fork of AFL. However, Python AFL does not instrument C code-based files. Hence, we built a script to validate and instrument C code-based files. To compare the performance of grey-box fuzzing against black-box fuzzing in the automotive system, we designed an unguided mutation fuzzer. The testing tool implements the mutation engine of VulFuzz and randomly attempts to find crashes in the system.

We build the framework in Python, and we execute all the experiments on the same machine with Intel Core i7-1065G7 processor, a four-core chip with Hyper-Threading that runs at a base frequency of 1.3GHz, and 8GB memory. The machine
Table 5.1: Comparative overview of VulFuzz, AFL, and the unguided mutation fuzzer running results

<table>
<thead>
<tr>
<th>Fuzzing Tool</th>
<th>Running Time</th>
<th>Num. of Test Cases</th>
<th>Num. of Dropped Cases</th>
<th>Num. of Conditional Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability-Oriented Fuzzer (VulFuzz)</td>
<td>16.8 Hours</td>
<td>1,810</td>
<td>20</td>
<td>4,812</td>
</tr>
<tr>
<td>American Fuzzy Lop (AFL)</td>
<td>8.7 Hours</td>
<td>1,002</td>
<td>233</td>
<td>4,809</td>
</tr>
<tr>
<td>Unguided Mutation Fuzzer</td>
<td>16.8 Hours</td>
<td>1,810</td>
<td>20</td>
<td>4,800</td>
</tr>
</tbody>
</table>

runs a 64-bit Ubuntu 16.04 Long Time Support (LTS) system.

5.2.2 Experiments and Results

We execute VulFuzz and AFL until they cannot discover new branches or reach vulnerable functions. Then, we run the unguided mutation fuzzer for the same number of test cases generated by VulFuzz. To test the performance of the proposed framework, we compare the number of test cases, dropped messages, coverage, and crashes, as shown in Table 5.1. To control the effect of randomness, we also report the results of 10 runs of each of the fuzz testing approaches. The results of the ablation study are discussed separately in the following subsection.

Test Case Analysis

As shown in Table 5.1, VulFuzz generates 1,810 test cases, 808 more test cases than AFL. The number of test cases affects the processing time. AFL finishes execution in half the time consumed by the other two fuzzers. As depicted in the framework of VulFuzz, weights are assigned for vulnerable functions to undergo several validations. Hence, even if a test case does not expand the coverage but evaluates vulnerable functions, it is preserved in the queue and further mutated. On the contrary, AFL
stores only the test cases that expand coverage. Thus, AFL requires fewer test cases to reach its goal.

**Dropped Test Case Analysis**

Next, we investigate the effectiveness of the mutation engine by examining the number of dropped messages of each testing tool. OpenPilot inputs include various data types, including Boolean, numerical, and string. As discussed in the depicted algorithm, the proposed mutation engine attempts to mutate the inputs with incompatible data types to validate the system’s input validation routine. Hence, generating 20 dropped messages. AFL’s mutation engine has remarkably more dropped messages than VulFuzz and the unguided mutation fuzzer. Specifically, out of the 1,002 generated test cases by AFL, 233 test cases do not pass OpenPilot’s input validation routine. That is, 23% of the test cases are dropped compared to 1% dropped by the other two testing tools. Automotive systems, like OpenPilot, have a stringent validation scheme, preventing random mutation from being an efficient method to validate the security of the system.

For example, Fig. 5.3 outlines a small part of a sample input of OpenPilot. To determine the vehicle’s health, the system takes *voltage* (numerical value), *ignition line* (boolean value), *controls allowed* (boolean value), *CAN send error* (numerical value), and *CAN forward error* (numerical value) as input. Seed s represents a good input used by the mutation engines to generate new seeds. The mutation engine of VulFuzz performs small mutations based on the input fields, resulting in two new inputs $s_1$ and $s_2$ that meet the criteria and help validate the system. AFL mutation engine performs a one-bit mutation changing ‘A’ from ‘FALSE’ to ‘@’ in $s_3$ and from
5.2. IMPLEMENTATION AND EVALUATION

Figure 5.3: Comparison between the vulnerability-oriented mutation engine and AFL’s mutation engine on a sample input

‘0’ to ‘p’ in $s_4$. Both new inputs $s_3$ and $s_4$ do not meet the input validation process of OpenPilot and are dropped.

AFL spends approximately 1.8 hours, over 20% of its processing time on invalid inputs. Hence, the proposed mutation engine outperforms small random mutation strategies and focuses on testing valid inputs capable of discovering vulnerabilities.

Coverage Analysis

To analyze VulFuzz coverage, we measure branch coverage and statement coverage. Branch coverage (also referred to as transition coverage) provides a better understanding of the fuzz testing capabilities in resolving complex conditions and reaching deeper code sections.

Table 5.1 presents the total number of visited conditional branches. The three
Figure 5.4: Comparison of statement coverage with respect to time in seconds (sec) approaches have relatively similar branch coverage reaching approximately 91% of the systems’ conditional branches. VulFuzz covers three more branches than AFL and 12 more branches than the unguided mutation fuzzer. As VulFuzz and AFL implement the same strategy to expand code coverage, their coverages are expected to be similar. Our approach achieved slightly better branch coverage due to the weights assigned to vulnerable functions. Mutating test cases that were not finding new branches but validating thoroughly weak functions eventually generated a seed input capable of discovering new branches.

We further study the testing tools’ coverage by analyzing the effect of weights on coverage behavior. Fig. 5.4 plots the statement coverage curves of each testing tool. We utilize the statement coverage in the analysis as it gives a broad vision of the coverage. AFL reaches its optimal coverage in 8.3 hours, while VulFuzz takes 15.8 hours. In the first 15.8 hours, VulFuzz prioritizes the search and evaluation toward
vulnerable components, not coverage expansion. Once comprehensive testing of high priority functions is completed, the fuzzer switches to low priority testing. Hence, after 15.8 hours, VulFuzz follows AFL’s approach. The main objective is to expand coverage at this stage, which is achieved quickly by the fuzzer since the test cases that help expand the coverage were saved in the low priority queue and not disregarded.

While AFL’s coverage plot and VulFuzz are similar in shape, the unguided mutation fuzzer has a different form. The fuzzer gradually reached its optimal coverage compared to a sharp increase in coverage in the other tools. This difference highlights the importance of testing guidance. The unguided mutation fuzzer attempts to validate the system randomly. Being unaware of the testing performance, the fuzzer cannot identify exceptional test cases that traverse the system. After wasting more than 11 hours looping around the same functionalities, the fuzzer randomly hits more statements.

Achieving good testing coverage for OpenPilot is not very challenging. Fig. 5.4 shows a sharp increase of the coverage in AFL and VulFuzz plots. Both fuzzers discovered test cases that run several large blocks of code. As a result, the prioritization role in the experiment is not well demonstrated. The regression test case suite prepared by Comma.ai aims to run all of OpenPilot’s subprocesses. Hence, with simple modifications, the three fuzzers can achieve sufficient coverage. Prioritization is efficient and needed when assessing the security of a full vehicle software system. The code size and complexity are amplified, making complete testing an unachievable target.
5.2. IMPLEMENTATION AND EVALUATION

Crash Analysis

Fig. 5.5 depicts the crashes triggered by the three testing tools. Crashes are exceptions raised by the automotive system due to unexpected behavior. The majority detected by the testing tools are index out of bound exceptions. For example, the system expects the radiator fan speed to be between 0 and 65,535 RPM. Any greater value causes the system to crash. Similarly, any temperature greater than 10,000 causes the system to crash. We have also found other types of vulnerabilities like the use of uninitialized values, value errors (illegal argument exceptions), and other unhandled exceptions.

As shown in the graph, the number of crashes identified by VulFuzz exceeds the crashes recognized by AFL and the unguided mutation fuzzer. The proposed framework achieves a total of 335 crashes. Fig. 5.5 shows a linear increase in the number
of discovered crashes by VulFuzz. Consistently, the proposed framework finds crashes during the first 15.8 hours of testing. At that time, the fuzzer was maintaining the test cases that traverse weighted functions. This is reflected in Fig. 5.4, with a steady coverage plot performing a thorough evaluation of weak functions. Out of 335 crashes, VulFuzz has 296 unique crashes\(^2\); 88% of the discovered crashes are unique. This shows the effectiveness of weight assignment in distributing the evaluation on different functions rather than randomly evaluating the system, which might block the search on a limited number of functions, resulting in fewer unique crashes.

The unguided mutation fuzzer attains a total of 176 crashes. However, only 45% of the obtained crashes are unique. The relatively high number of duplicate crashes identified by the unguided mutation fuzzer shows the ineffectiveness of black-box fuzz testing in discovering various code regions. In general, the mutation engine and the number of generated test cases heightened the testing tool’s performance and enabled it to find more crashes than AFL. We intentionally ran the random fuzzer for 1,810 test cases to assess the importance of grey-box testing in the vehicle industry. This gives the fuzzer a fair chance to find crashes. Still, VulFuzz discovers 90% more crashes and 274% more unique crashes than this black-box testing method. The effectiveness of the proposed mutation engine certainly boosted the performance of black-box validation. The fuzzer did not consume time on invalid input; 99% of the test runs were successful. A random black-box fuzz testing technique would have less effective results, attempting to create arbitrary inputs not accepted by automotive systems.

AFL has poor performance in terms of discovered crashes. In the first 4.5 hours,

\(^2\)Following the AFL definition of unique crashes, we consider a crash unique if its execution trace involves at least one new state transition not detected in previously discovered crashes.
AFL detects eight crashes which are all unique crashes. As discussed earlier, AFL’s mutation engine has a remarkable performance while working on media files. However, it is less efficient with a complex system that incorporates a robust input validation mechanism. Testing hours are lost on invalid inputs that do not evaluate the system and seek crash identification. AFL achieves its coverage peak relatively quickly. Nevertheless, this affects the number of detected crashes. As shown in Fig. 5.4, during the first 4.5 hours, the fuzzer was still attempting to expand coverage but also examining vulnerable functions. Experimenting with the new test cases helped AFL trigger undiscovered crashes. Once AFL increases coverage, fewer crashes are discovered.

We also investigate the relationship between weighted functions and unique crashes. Fig. 5.6 compares the number of unique crashes to the number of times weak functions are tested for the three testing tools. VulFuzz uses security vulnerability metrics to identify the system’s weak components. A thorough evaluation of these components is achieved by assigning weights. Based on the security vulnerability scores, functions in OpenPilot were classified as low (functions are not vulnerable), medium (functions may pose some risk), and high (functions are vulnerable). Functions with low vulnerability were assigned a weight of value 0 to avoid prioritizing the testing toward components that pose minimal risk to the system. Medium vulnerability functions are assigned a weight of value 50, and high vulnerability functions are assigned a weight of value 100. 5% of OpenPilot’s functions have a high number of vulnerabilities, and 3% of the functions have a medium number of vulnerabilities. VulFuzz generates 808 test cases that examine the high and medium vulnerability functions compared to 188 test cases for the unguided mutation fuzzer and 79 test
cases for AFL. As shown in Fig. 5.6, the more the vulnerable functions are tested, the higher the number of discovered unique crashes. AFL has the lowest execution count of weighted functions, which is reflected in the number of discovered crashes. In comparison, VulFuzz’s exhaustive evaluation of vulnerable components enhanced its crash detection power. This confirms the importance of security metrics and weight assignments. The security metrics direct the testing toward complex functions that are more prone to bugs. The weight assignment gives VulFuzz a chance to examine these components more and identify vulnerabilities.

Fig. 5.7 compares the three testing tools’ reported unique crashes. VulFuzz identifies all the crashes recognized by AFL and 59 of the unique crashes detected by the unguided mutation-based fuzzer. The proposed framework does not identify only 12 of the unique crashes found by the mutation-based fuzzer (4% of the total).
We further examine the effectiveness of VulFuzz in identifying unique crashes by running the framework on different subprocesses. As discussed earlier, OpenPilot is a large system composed of different subprocesses, and we performed the evaluation on the whole system. However, to show that VulFuzz consistently recognizes crashes on vehicle software components, we run the framework sequentially on three of OpenPilot’s subprocesses and compare the recognized unique crashes with the results obtained by the other testing tools. The processes were chosen randomly. The results presented in Table 5.2 confirm the effectiveness of VulFuzz in identifying unique crashes of vehicle software components while consistently outperforming AFL and the unguided mutation fuzzer that do not target the automotive industry testing challenges. For example, for subprocess 1 (Controls), VulFuzz recognizes 106% more unique crashes than the unguided mutation fuzzer and 675% more than AFL. Similarly, for subprocess 2, VulFuzz detects 150% more unique crashes than AFL and
5.2. IMPLEMENTATION AND EVALUATION

Table 5.2: Crash analysis on different OpenPilot subprocesses

<table>
<thead>
<tr>
<th>OpenPilot Subprocesses</th>
<th>Num. of Unique Crashes by VulFuzz</th>
<th>Num. of Unique Crashes by AFL</th>
<th>Num. of Unique Crashes by Unguided Mutation Fuzzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subprocess 1 (Controls)</td>
<td>31</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>Subprocess 2 (Radar)</td>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Subprocess 3 (Calibration)</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.3: Average results of 10 runs

<table>
<thead>
<tr>
<th>Fuzzing Tool</th>
<th>Running Time</th>
<th>Num. of Test Cases</th>
<th>Num. of Dropped Cases</th>
<th>Num. of Unique Crashes</th>
<th>Num. of Conditional Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability-Oriented Fuzzer (VulFuzz)</td>
<td>16.7 Hours</td>
<td>1,810</td>
<td>20</td>
<td>319</td>
<td>4,811</td>
</tr>
<tr>
<td>American Fuzzy Lop (AFL)</td>
<td>8.4 Hours</td>
<td>1,018</td>
<td>153</td>
<td>12</td>
<td>4,800</td>
</tr>
<tr>
<td>Unguided Mutation Fuzzer</td>
<td>16.6 Hours</td>
<td>1,810</td>
<td>19</td>
<td>80</td>
<td>4,795</td>
</tr>
</tbody>
</table>

obtains similar results with the unguided mutation fuzzer.

**Average Run**

To mitigate the randomness’s effect on the evaluation process results, we run each fuzzing tool ten times and calculate the average results. The average results are presented in Table 5.3. We notice that the randomness incorporated in the AFL mutation engine and VulFuzz mutation engine has a minor impact on the performance of the fuzzing tools. The results of the ten runs are very similar. On the contrary, the lack of guidance and the randomness factor affect the unguided-mutation fuzzer coverage results. Six out of the ten runs cover less than 4,790 conditional statements. Hence, grey-box fuzz testing is vital to maintain code coverage and efficient evaluation.
5.2. IMPLEMENTATION AND EVALUATION

Table 5.4: Results of the ablation study on VulFuzz engines

<table>
<thead>
<tr>
<th>Eliminated Engine</th>
<th>Running Time</th>
<th>Num. of Test Cases</th>
<th>Num. of Dropped Cases</th>
<th>Num. of Unique Crashes</th>
<th>Num. of Conditional Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>None Removed</td>
<td>16.8 Hours</td>
<td>1,810</td>
<td>20</td>
<td>296</td>
<td>4,812</td>
</tr>
<tr>
<td>Vulnerability</td>
<td>2 Hours</td>
<td>341</td>
<td>18</td>
<td>59</td>
<td>4,818</td>
</tr>
<tr>
<td>Evaluation</td>
<td>14.35 Hours</td>
<td>1,600</td>
<td>19</td>
<td>290</td>
<td>4,811</td>
</tr>
<tr>
<td>Mutation</td>
<td>21.2 Hours</td>
<td>2,367</td>
<td>702</td>
<td>50</td>
<td>4,797</td>
</tr>
<tr>
<td>Prioritization</td>
<td>16.9 Hours</td>
<td>1,820</td>
<td>20</td>
<td>338</td>
<td>4,812</td>
</tr>
</tbody>
</table>

5.2.3 Ablation Study

Each engine deployed within VulFuzz aims to intensify the vulnerability identification process of vehicle software systems. This section performs an ablation study to demonstrate each engine’s functionalities and their contribution to vulnerability exposure. We run VulFuzz four times, and in each run, we eliminate one of the four engines. The results of the ablation study are summarized in Table 5.4.

The vulnerability engine plays an essential role in orienting the fuzz testing toward weak components that requires intensive testing. Omitting the vulnerability engine affects other engines of the framework. The evaluation engine solely monitors seed execution and updates the coverage table without monitoring the weight count. Moreover, test case prioritization relies on the vulnerability score. Hence, by eliminating the vulnerability engine, the prioritization engine role is excluded too. Removing the vulnerability engine negatively affects the number of discovered crashes. Only 59 unique crashes were located compared to 296 crashes by VulFuzz. With the mutation engine’s help, the fuzz testing achieves good coverage with only 341 test cases. This limits the testing time and the intensive testing of weak components forced by the vulnerability engine.

In the second run of the ablation study, we limited the evaluation engine’s role to
update the call count of weight functions only. Hence, with this change, the fuzz testing does not recognize test cases that traverse new branches and halts after satisfying the call count of weighted functions. Limiting the evaluation engine functionalities does not drastically affect the fuzz testing results. The vulnerability engine forces a thorough examination of the weak functions, which resulted in a high number of test cases and the discovery of 290 unique crashes.

To validate the mutation engine’s critical role, in the third run of this study, we replace the VulFuzz mutation engine with AFL’s mutation engine. Eliminating the VulFuzz mutation engine increases the number of dropped messages considerably to 702 dropped test cases. This, in turn, affects the fuzz testing progress in identifying unexpected behavior. Only 50 unique crashes were discovered compared to 296 by VulFuzz. The fuzz testing in the third run failed to find enough test cases that thoroughly examine the weighted functions. Still, eliminating the mutation engine resulted in fuzz testing that is more performant than AFL. Prioritizing the testing toward vulnerable functions helped to discover 525% more unique crashes than AFL.

The final step of the ablation study omits the prioritization engine from VulFuzz. Prioritizing the test case is valuable in the vehicle industry when the security testing time is insufficient to perform a comprehensive evaluation. However, when performing complete testing, the prioritization engine does not enhance the results. As shown in Table 5.4, the fourth run has similar results to VulFuzz. 338 unique crashes were discovered by generating 1,820 test cases.

In conclusion, the aggregation of engines makes VulFuzz a powerful fuzz testing tool; eliminating any of them makes VulFuzz less effective. However, the vulnerability and the mutation engines are the most critical ones that play an essential role in
identifying vulnerabilities within vehicle software systems efficiently and comprehensively.

5.3 Summary

This chapter proposes a vulnerability-oriented grey-box fuzz testing framework that overcomes black-box testing limitations by acquiring some knowledge about the system without causing the overhead that white-box testing causes. VulFuzz is specifically tailored to test and identify vehicle software systems’ likely vulnerabilities. The framework incorporates four engines: (1) Vulnerability, (2) Mutation, (3) Evaluation, and (4) Prioritization that cooperate to guide the testing toward the system’s vulnerabilities. In contrast to black-box fuzzers that blindly verify the system, the proposed framework utilizes security metrics to supervise and guide the testing. The security metrics quantitatively measure the vulnerability of components within a vehicle software system. Such an estimation reflects the code complexity and identifies the weak integration that can be violated by an attacker. According to the vulnerability value, each component is assigned a weight, representing the number of times a component should be tested. A thorough examination of potentially weak components can boost vulnerability detection and assure a secure system. VulFuzz monitors the coverage of seed inputs to achieve its goal and prioritize the testing. To strengthen the grey-box fuzzer performance, the mutation engine generates various test cases that comply with the automotive system’s input structure by inferring the inputs’ data types.

To evaluate the effectiveness of the proposed framework, we experiment on an automotive system, OpenPilot. VulFuzz outperforms a black-box fuzzer and the famous grey-box fuzzer, AFL. Thorough testing of complex components of OpenPilot
helped VulFuzz find 335 crashes, 4088% more than AFL and 90% more than unguided mutation fuzzer in 16.8 hours (100% more than AFL and 0% more than unguided mutation fuzzer). Moreover, the framework’s data type-based mutation satisfied the input requirements of OpenPilot, with only 20 dropped messages, which is 91% less than AFL.

These promising results show the effectiveness of VulFuzz in exposing vehicle software systems vulnerabilities. The framework offers a reliable security testing tool that does not increase testing complexity but intelligently and efficiently identifies weak components to focus on them. Moreover, prioritizing the testing can aid security engineers in managing the security testing in time-limited projects automatically. Nevertheless, grey-box fuzz testing might fail to reach some functionalities in the system. For safety-critical components, it is vital to support the fuzz testing with another tool that complements it and assures full testing coverage. In the next chapter (Chapter 6), we introduce a hybrid fuzz testing framework that boosts the grey-box fuzzing performance and explore the deep paths of automotive systems.
Chapter 6

Hybrid Fuzz Testing for Vehicle Systems

Grey-box fuzz testing [22, 107–111, 169] validates automotive systems while acquiring some knowledge about the system. As discussed in the previous chapter, such a testing method can run automatically without increasing the testing complexity, promising efficient evaluation of automotive systems. Nevertheless, like black-box fuzzers, grey-box fuzzers fail to evaluate deep paths with complex checks, potentially leaving many vulnerabilities undiscovered. For this reason, it is vital to complement grey-box testing with another testing method that guarantees a comprehensive and deep evaluation of weak components.

In general, to overcome the limitations of black-box and grey-box fuzzing, researchers employ symbolic execution engines to validate all possible control flow paths [112]. However, symbolic execution creates severe challenges (e.g., path explosion and memory overhead) that make it impractical for complex and large systems like automotive systems. To balance between black-box and symbolic execution, some researchers utilize concolic testing, which is known as white-box fuzz testing [170]. Concolic testing is a hybrid testing approach that combines concrete and symbolic evaluation to discover new branches within an execution. Concrete inputs drive the
testing towards a particular path, which is explored symbolically as well. Neverthe-
less, leading the testing with few random concrete inputs is not enough to reduce the
testing time and explore automotive systems.

The objective of this research is to design a prioritized and targeted concolic
engine that complements the grey-box fuzzer. The targeted concolic engine selectively
utilizes symbolic execution on functions that can expand the vulnerability exposure
and reach deeper code regions. Hence, it expands the coverage without increasing the
complexity of the testing.

This chapter presents VulFuzz++, a hybrid fuzz testing framework that starts by
executing the grey-box fuzzer, VulFuzz, which is introduced in Chapter 5. When Vul-
Fuzz fails to identify new seed inputs that can explore untraversed paths, it halts. At
this stage, VulFuzz++ analyzes the coverage results and prioritizes uncovered branches
giving higher importance to the weakest components that are more prone to attacks.
Next, new inputs are generated using a targeted concolic engine that operates with
optimal inputs to reduce the exploration time. Finally, the newly formed inputs are
passed again to the grey-box fuzzer to evaluate the target region thoroughly. Hence,
restricting the use of symbolic execution empowers VulFuzz++ to explore deeper and
vulnerable paths without causing significant overhead.

We start this chapter by illustrating the need to expand grey-box fuzz testing
and introduce symbolic and concolic execution. We present the VulFuzz++ framework
and provide an overview of its stages. Next, we discuss the three stages of VulFuzz:
grey-box fuzzing, prioritized and targeted concolic execution, and targeted grey-box
fuzzing. Finally, we discuss the experiments and results showing the effectiveness of
VulFuzz++. 
6.1 Background and Motivation

This section discusses the shortcomings of grey-box fuzz testing and highlights the need for supporting similar fuzzers with symbolic execution. It provides background knowledge about symbolic and concolic execution, demonstrating their capabilities in satisfying input-specific conditions.

6.1.1 Limitation of Grey-box Fuzzing

Black-box and grey-box fuzzers efficiency is grounded on requiring no or minimal knowledge of the system. As this type of fuzzer does not analyze the code, input generation follows a naive approach that blindly creates seeds, hoping they pass the system checks. The grey-box fuzzer VulFuzz employs a data type-based mutation engine that applies mutations at the inputs’ high-level design. VulFuzz mutation engine bypasses autonomous system input validation by using a dictionary to preserve the input format. Thus, the engine assures that the testing can pass the system’s elementary checks. However, random mutation cannot satisfy input-specific branch conditions, preventing the fuzzer from exploring deeper paths of the automotive system. Consider the example shown in Listing 6.1, which has one vulnerable function, `errorNotification`. VulFuzz mutation engine can promptly satisfy the first conditional statement that requires the `thermal` variable to be a non-empty string. However, it is improbable to satisfy the second condition that requires the `thermal` value to be precisely ‘ChargingError’ by a random data type-based mutation. Hence, black-box and grey-box fuzzers, in similar cases, fail to validate the automotive system thoroughly.
6.2 Symbolic and Concolic Execution

Unlike black-box and grey-box fuzzers, symbolic execution can easily find an input that satisfies the second condition of Listing 6.1 and traverses the vulnerable function. During execution, the engine maintains symbolic formulas for each explored flow graph and controls a symbolic memory that associates each variable with its symbolic expression. Then, collected conditions are negated and solved using Satisfiability Modulo Theories (SMT) solvers to generate new inputs. Such testing techniques perform a thorough verification for a system. It assures the validation of complex conditional statements that are hard to satisfy with random mutation. While validating all paths of automotive components can assure high security, symbolic execution suffers from significant shortcomings that make it incapable and inefficient to validate automotive systems.

```python
def tempBattery(thermal, battery):
    if thermal != None:  # first condition
        if battery < 1 and thermal == 'ChargingError':  # second condition
            errorNotification()  # VULNERABLE

def main():
    read(thermal, battery)
    tempBattery(thermal, battery)
```

Listing 6.1: An example showing an input-specific condition

Concolic testing executes a program both concretely and symbolically. A sample concrete input directs the execution towards a particular path. At the same time, the symbolic engine represents the inputs symbolically and collects symbolic constraints along the execution path. Then, one of the constraints is negated and solved using an SMT solver. The newly generated input then leads the execution again. This procedure is repeated until all possible paths are examined. Thus, concolic testing limits the path explosion problem. Still, few random concrete inputs cannot address
the symbolic execution critical limitations. In the following paragraphs, we discuss the notable shortcomings of symbolic and concolic executions.

**Memory Overhead.** Traditional concolic engines store the execution state and symbolic variables of an execution path in memory to be analyzed and processed by SMT solvers. As a result, memory consumption caused by symbolic exploration can be remarkably high for complex and large systems that embed arrays, pointers, and other complex structures. Memory is a scarce resource for automotive software that runs on Electronic Control Units (ECUs).

**State Space Explosion.** Typically, the concolic engine attempts to execute the full path of multiple concrete inputs symbolically. This evaluation suffers from the path explosion problem; the number of execution states that need to be symbolically traversed is very high, resulting in an immense search space. The complexity of automotive systems increases the execution states, causing critical path explosion problems.

**Expensive Constraint Solving.** SMT solvers are capable of solving complex conditions that utilize multiple variables. However, they are impractical when dealing with non-linear arithmetic conditions, spending hours of testing time striving to solve the desired condition.

**External Environment.** Systems calls and calls to libraries form a bottleneck for concolic engines. The symbolic executor cannot interpret the behavior and functionalities of environmental calls as they are not part of the system being tested.

**Time Consuming.** In contrast to concrete execution that runs on a specific input that explores a single path, concolic engines perform multiple tasks beyond simple exploration. For example, they collect conditional branches, save execution traces,
symbolically represent formulas, and solve branch constraints. Such a thorough ex-
amination is time-consuming. As a result, the feasibility of such a test becomes
questionable in time-sensitive industries like the automotive industry.

VulFuzz++ is designed to consider all these challenges, strategically employing
solutions that mitigate the shortcomings while benefiting from the power of symbolic
execution to find input-specific seed cases.

6.3 A Hybrid Fuzzing Framework for Vehicles

The VulFuzz++ framework offers a robust and trustworthy testing tool that main-
tains the performance of fuzz testing while utilizing prioritized and targeted concolic
exploration. VulFuzz++ concolic engine limits the symbolic exploration to only a few
targets that are capable of increasing the coverage and expanding the vulnerability
identification.

VulFuzz++ has three stages depicted in Fig. 6.1. The framework strives to run
one type of testing as long as possible to minimize the memory consumption needed
when switching between different modes. The first stage runs the grey-box fuzzer
VulFuzz, introduced in Chapter 5, extensively to explore the automotive system and
detect vulnerabilities efficiently. The mutation engine of VulFuzz plays a vital role
in eliminating the need for continuous interweaving between the two testing types.
VulFuzz data type mutation engine assures that the testing will not be stuck at the
initial validation criteria of automotive systems. Hence, since the fuzzer can evaluate
most paths that are not input-specific, symbolic exploration intervention is not needed
in the early stages of the testing. The second stage employs a customized concolic
ingine that generates new inputs capable of exploring new deeper paths. This stage

executes with multiple paths that examine different code regions, enabling the testing to achieve near-optimal soundness with the help of SMT solvers. Finally, the last stage runs a modified version of the grey-box fuzzer that thoroughly validates the newly discovered paths and their surrounding region. Stages two and three can run two times depending on the identified priorities.

Each stage of the VulFuzz++ framework performs some steps that help the hybrid fuzz testing tool achieve its goal. The steps and the stages are shown in Fig. 6.1 and are listed as follows. The first stage, grey-box fuzzing, is composed of four steps. Steps 1, and 2 prepare for the testing at compilation time. Both of these steps are needed by the grey-box fuzzer in stage one and the concolic engine in stage two.

1. Using the automotive system source code, VulFuzz++ starts by calculating the security vulnerability of different components. Five security vulnerability metrics are used at this step to identify the weakest components of the system.

2. The call graph of the system is generated, which is used by the fuzzers to monitor the progress of testing.

3. The guided grey-box fuzzer then begins exploring the system and identifying vulnerabilities.

4. Once VulFuzz halts, the framework updates the list of seed inputs used to evaluate the automotive components, the call count of weighted functions, and the path trace and coverage tables.

Stage two, prioritized and targeted concolic execution, then begins.

5. Using the call count of weighted functions and the path traces, VulFuzz++ recognizes and prioritizes the untraversed nodes, giving higher priority for nodes that are vulnerable and connected to various untraversed nodes.

6. Next, starting with the high priority nodes, the grey-box fuzzer seed inputs are analyzed, and VulFuzz++ selects the optimal seed inputs.

7. To pave the road for effective symbolic execution, VulFuzz++ identifies
Figure 6.1: The VulFuzz++ framework composed of three stages

the functions that need to be symbolically traversed. The hybrid testing tool next invokes its targeted concolic engine attempting to generate inputs that satisfy the conditions of untraversed nodes. The optimal inputs drive the execution, and the
target functions are executed and traced symbolically. Using SMT solver, new seed inputs are generated that can expand the coverage.

In stage three, the execution is transferred again to the fuzzer. A directed version of VulFuzz uses the newly generated seed inputs to thoroughly evaluate the target nodes and their untraversed connected paths. Finally, when the directed VulFuzz terminates if stage three is running for the first time, the untraversed nodes are reviewed and updated. This step reviews whether stage three fuzzing achieved traversing a node that belongs to the low priority queue to be removed. It also identifies whether the fuzzing progressed in approaching a vulnerable untraversed node to join the low priority queue. Then steps to are repeated for the updated nodes of the low priority queue. When the directed VulFuzz executes the low priority nodes, the framework halts marking the end of the security testing.

6.4 Stage One: Grey-box Fuzzing

Within a large system like an automotive system, not all components pose the same security risk. VulFuzz [22] implements four engines that cooperatively direct and prioritize the testing towards the weakest components of the automotive system. VulFuzz strives for a thorough evaluation of vulnerable components while expanding coverage. VulFuzz engines are discussed in detail in Chapter 5.

6.5 Stage Two: Prioritized and Targeted Concolic Execution

When the fuzzer halts, failing to explore new transitions that expand the coverage, VulFuzz++ starts stage two. The prioritized and targeted concolic execution stage aims to leverage the strength of symbolic exploration in finding inputs that drive the
testing to new paths. As discussed in Section 6.1, symbolic execution has critical shortcomings that affect the performance of security testing. As VulFuzz++ strives to offer an efficient security verification tool for the automotive industry, it employs a tailored concolic engine that diminishes the drawbacks while maintaining durability. The core idea is to limit symbolic exploration only to specific functions.

Traditional concolic engines used in hybrid fuzzing tools symbolically trace the entire execution path of concrete inputs. These engines spend a considerable amount of time with high memory consumption to identify state transitions that were not discovered by the fuzzing engines. This requires tracking all concrete and symbolic values in memory and registers during the execution. In addition, standard concolic engines perform constraint resolution for the gathered conditions multiple times throughout an execution, attempting to find new inputs.

Consider the example shown in Listing 6.2. A traditional concolic engine requires any seed input to lead the execution. For example, a seed with \( \text{thermal} = \text{'green'} \) and \( \text{battery} = 10 \) drives the concolic execution. All the paths representing the processing of this input are symbolically analyzed, including function \( \text{lowBattery} \) that does not have any input-specific condition and is evaluated easily with fuzz testing. Though Listing 6.2 only has two functions that do not cause severe overhead, in reality, the concolic execution will have to explore many more paths in the same manner.

In contrast to existing hybrid fuzzing approaches that depend on the concolic engine to identify new input, VulFuzz++ offloads this job from its concolic engine and uses it to gather symbolic representations only. New targets are specified using the vulnerability scores, coverage table, and trace logs of the inputs generated by VulFuzz. For the example shown in Listing 6.2, VulFuzz++ identifies the second condition
of tempBattery as a target since it was not satisfied. Then it finds the optimal input (e.g., thermal = ‘green’ and battery = 10). All the paths execute concretely except for the paths that belong to tempBattery execute symbolically. The gathered conditions are negated, and the SMT generates an input with thermal value equal to ‘ChargingError’, allowing VulFuzz++ to explore the vulnerable process errorNotification. Hence, focusing only on relevant constraints allows VulFuzz++ to produce new test cases that explore different code paths with minimum overhead.

```python
def lowBattery(battery):
    if battery < 12:
        chargeSignal()  # NOT VULNERABLE

def tempBattery(thermal, battery):
    if thermal != None:  # first condition
        if battery < 1 and thermal == 'ChargingError':  # second condition
            errorNotification()  # VULNERABLE

def main():
    read(thermal, battery)
    lowBattery(battery)
    tempBattery(thermal, battery)
```

Listing 6.2: An example illustrating the usefulness of targeted symbolic exploration

### 6.5.1 Shifting to Concolic Execution

With a more complex structure than Listing 6.2, shifting to concrete execution becomes more challenging. VulFuzz++ achieves its goal by preparing for targeted concolic execution. Steps 5 to 7 of the framework are dedicated to accomplishing this job. The details of the steps are described in what follows.

**Prioritize Untraversed Branches.** The path trace and coverage table generated by VulFuzz shows the fuzzer performance and coverage information. Using the call graph generated from the source code in step 2, the nodes with untraversed conditions are hierarchically explored in a top-down fashion. The call graph (CG) of a
component ($C$) has a set of Nodes ($N$) representing the functions of $C$. The calls between functions are represented by a directed edge between the nodes ($n \rightarrow n'$). The nodes are prioritized according to the objective of examining vulnerable functions and expanding code coverage. As concrete inputs drive the concolic execution, only the branches that belong to a traversed function can be prioritized. Since VulFuzz++ runs the grey-box fuzzer quite exhaustively in stage one, the number of explored paths is expected to be high.

Vulnerable nodes with untraversed branches and non-vulnerable nodes with a path to a weak untraversed function pose a high risk to the automotive system. Hence, they are added to the high priority queue. Moreover, nodes with untraversed branches connected to several untraversed paths are also placed in the high priority queue. When the fuzzer is given an input that traverses a specific node $n$, simple mutations can help explore multiple paths that traverse node $n$. This is because node $n$ can have an input-specific condition prohibiting the execution from exploring deeper paths. For example, once the second condition in Listing 6.2 is satisfied with an input that set the `battery` value to less than 1 and `thermal` value to ‘ChargingError,’ the execution is driven to function `errorNotification`, which can expand the coverage and explore new paths.

Nodes with untraversed branches connected to a node already in the high priority queue are added to the low priority queue. VulFuzz++ avoids symbolic exploration for these nodes in the first run since they may be explored easily by the fuzzer (untraversed nodes are identified following a top-down approach). However, if the fuzzer fails to explore them (e.g., they are also input-specific nodes), they are targeted in the second run of stage two. Nodes with untraversed branches that are not connected
6.5. STAGE 2: PRIORITIZED & TARGETED CONCOLIC EXECUTION

to many untraversed paths join the low priority queue. More priority queues and iterations between stages two and three can be created depending on the automotive component being tested and the testing goals.

**Select Optimal Inputs.** Typical hybrid fuzz testing techniques choose seed inputs to drive the testing based on their coverage performance [119]. Such techniques do not have any specific target to resolve and depend on the concolic engine to find new paths. On the contrary, VulFuzz++ attempts to resolve nodes that belong to one of the queues. To reduce the number of symbolically executed functions, VulFuzz++ runs the symbolic execution with optimal inputs. We define an optimal input as *an input that traverses the highest number of targeted nodes*. For example, consider seeds $s_1$ and $s_2$. Seed $s_1$ traverses target nodes $n_1$ and $n_2$, while seed $s_2$ execution path covers target nodes $n_1$, $n_2$, and $n_3$. Seed $s_2$ is considered an optimal input, and seed $s_1$ is not used to direct the concolic execution. If more than one input with the same number of traversed targeted nodes exists, we utilize the input with the highest coverage. The selection of optimal inputs is concurrently performed for all the nodes to reduce the overhead and multiple traversals.

**Identify Target Functions.** The example shown in Listing 6.2 requires the function `tempBattery` to be executed symbolically to resolve the second condition and execute the vulnerable function `errorNotification`. The variable `thermal` is a user input that was not manipulated by any other function. While some automotive inputs follow the same structure, many untraversed conditions comprise variables that are not direct inputs. Consider the example in Listing 6.3, with one vulnerable function `lowBattery`. Traversing the vulnerable region requires the value of `battery_percent` to be 20. Symbolic analysis of function `lowBattery` can identify that the value of
battery_percent has to be 20 to increase the coverage. However, battery_percent is not a user input, only battery value is. The relationship between battery and battery_percent is unknown to the targeted concolic engine, failing to generate a test case that traverses the vulnerable code region. Even though such a scenario is also common in traditional concolic engines [112], as VulFuzz++ performs targeted concolic exploration and does not symbolically explore the complete input processing path, encountering such a scenario becomes more probable.

```python
def lowBattery(battery_percent):
    if battery_percent == 20:
        ... #VULNERABLE

def voltagePercent(battery):
    ...
    if battery >= 11.58 and battery <= 11.90:
        battery_percent = 20
    ...

def main():
    read(battery)
    battery_percent = voltagePercent(battery)
    lowBattery(battery_percent)
```

Listing 6.3: An example that requires symbolic exploration for multiple functions

To augment the vulnerability exposure process, VulFuzz++ limits these scenarios by identifying the relationship between functions and variables. For a target node $n_1$, the unexplored conditions are identified and parsed. Then, the variables utilized in these conditions are analyzed. If they are related to user input, then targeted symbolic exploration does not require any further information. The node $n_1$ joins the list of targeted functions. Otherwise, all the functions that manipulate the variables in the conditions are recognized using trace information and code parsing. If node $n_1$ utilizes a variable manipulated by a parent node $n_0$, then node $n_0$ is added as a target function.
For the example in Listing 6.3, using code parsing VulFuzz++ identifies that the `voltagePercent` function is adjusting the `battery_percent` variable. Hence, `voltagePercent` is added as a target function, and the concolic engine symbolically represents its conditional branches. The SMT solver identifies an input (e.g., `battery = 11.58`) that sets the `battery_percent` variable to 20, executing the vulnerable code region without requiring explicit symbolic exploration for `lowBattery` function.

As we follow a hierarchical top-down approach in function discovery, the occurrence of similar cases is not very high. When the untraversed nodes are prioritized, the function `voltagePercent` is identified as a high priority node connected to a vulnerable function (`lowBattery`) with untraversed branches.

### 6.5.2 Tailored Concolic Engine

VulFuzz++ uses a concolic engine tailored to accommodate the automotive industry needs. Inspired by selective symbolic execution [171], a technique that switches between concrete and symbolic exploration, the designed concolic engine is driven by concrete inputs and symbolically executes only a limited number of functions along the execution path. After identifying the target functions, the concolic engine can start. For every optimal input, VulFuzz++ concolic engine traces the execution following the paths of the concrete input. When the execution reaches a targeted function, VulFuzz++ concolic engine starts the symbolic exploration, all non-constant variables are symbolically represented, and conditional branches are represented as symbolic constraints. When the concrete inputs force the execution to a non-targeted function, the symbolic exploration stops.

**Symbolic Exploration of Function Calls.** When a targeted function calls another
function, and the concrete inputs drive the execution towards these functions, then VulFuzz++ concolic engine explores the called functions symbolically as well. This is because the called functions can affect the constraint solving process. Consider the example in Listing 6.4. If VulFuzz++ concolic engine is symbolically exploring function lowBattery, the SMT solver fails to find an input that satisfies the first condition of lowBattery without symbolically exploring voltagePercent. Hence, to increase the chances of finding new inputs, VulFuzz++ concolic engine explores the called functions symbolically.

```python
def lowBattery(battery):
    battery_percent = voltagePercent(battery)
    if battery_percent == 20: #first condition
        ... #VULNERABLE

def voltagePercent(battery)
    ...
    if battery >= 11.58 and battery <= 11.90:
        battery_percent = 20
    ...

def main():
    read(battery)
    lowBattery(battery)
```

Listing 6.4: An example showing function calls

**Overcoming Symbolic Exploration Shortcomings.** VulFuzz++ concolic engine diminishes the drawbacks of symbolic exploration by guiding the testing with concrete inputs, which limits the path explosion problem. On top of that, VulFuzz++ employs targeted symbolic exploration for only a few functions, lessening state space explosion and saving execution time. Though function calls for a target function are symbolically explored, if nesting calls exceed a certain number, VulFuzz++ concolic engine switches to concrete execution. Moreover, similar to other concolic engines [119], VulFuzz++ supports external environmental calls, including system calls and external library calls. The framework focuses on the heavily used external calls like integers
operators and string operators. For unsupported environmental calls, VulFuzz++ con-
colic engine uses concrete execution. Finally, VulFuzz++ prioritizes the branches and
finds an optimal input to traverse the largest number of functions simultaneously,
minimizing repetitive exploration and limiting memory consumption.

6.6 Stage Three: Targeted Grey-box Fuzzing

Stage three employs fuzz testing to thoroughly evaluate the nodes discovered in
stage two and their surrounding region. Rerunning VulFuzz without any modifica-
tions wastes hours of testing time striving to traverse new branches, which are hard to
target. The fuzzer already attempted to satisfy the untraversed branches in the first
stage but failed. We design a modified version of VulFuzz that minimizes the testing
time by generating test cases that avoid traversing code regions already evaluated
and direct the testing towards the nodes found in stage two. The directed version
of VulFuzz used in stage three introduces some modifications to the mutation, eval-
uation, and prioritization engines of VulFuzz, detailed as follows. The vulnerability
engine of VulFuzz runs at compilation time, and its results are not altered during
the security testing. Thus, the stage three fuzzer uses the vulnerability information
already obtained by VulFuzz.

Directed VulFuzz Mutation Engine. To ensure a thorough examination of the
paths discovered in stage two, stage three fuzzer generates new inputs that traverse
these paths. For each new seed file, the fields that were satisfied by the SMT solver
are marked. Stage three modifies the data type-based mutation of VulFuzz. Instead
of randomly mutating any field to generate a new test case, the directed VulFuzz an-
alyzes the seed file and chooses a field to mutate. Using the engine’s dictionary, stage
three fuzzer identifies the fields that belong to the same component of the marked field and chooses one random identified field to mutate using data type operations. Directed VulFuzz does not mutate the marked fields to assure coverage of the newly discovered nodes.

**Directed VulFuzz Evaluation Engine.** To detect whether a test case achieves the fuzzing goal, the evaluation engine monitors the test case executions. It identifies the test cases that traverse stage two nodes or discover a new branch connected to those nodes. Similar to the VulFuzz engine, the directed version of the fuzzer’s evaluation engine also records the test cases to evaluate vulnerable functions.

**Directed VulFuzz Prioritization Engine.** Since stage three runs two times, one for high priority nodes and another for the low priority nodes, the prioritization engine role of the directed VulFuzz is restricted to assigning energy values. Following the same methodology of VulFuzz, the stage three prioritization engine adopts a constant energy assignment while giving higher importance to some test cases. Test cases that evaluate vulnerable functions or have a path to one are assigned triple the energy value of other inputs. Test cases that discover a new branch connected to several untraversed branches are assigned double the energy value. Higher energy values give the mutation engine a better chance to increase the coverage of the surrounding nodes of stage two.

### 6.7 Evaluation

To assess the effectiveness of VulFuzz++ in augmenting coverage and discovering new vulnerabilities, we implement the framework and evaluate its performance on an automotive system. This section starts by describing the implementation details.
Then it assesses the performance of VulFuzz++ and compares it with other fuzzers utilized in the automotive industry.

6.7.1 Implementation

The VulFuzz++ framework is built in Python. It utilizes the grey-box fuzzer VulFuzz and modifies it to satisfy the requirements of the fuzzer used in stage three. VulFuzz++ concolic engine is developed using existing concolic libraries [162] while employing certain modifications to cater for targeted symbolic exploration. It employs a tracer to obtain the coverage arcs and collect the conditional constraint that the execution path encounters for the target functions. To solve the constraints, we employ Z3 [172] an efficient open-source Satisfiability Modulo Theories (SMT) solver that supports different theories, including arithmetic, algebraic data types, sequences, and strings.

We evaluate the performance of VulFuzz++ on an open-source autonomous driving and safety assistant system, OpenPilot (Version 0.7.8) [156]. The automotive system is designed using both Python and C languages. As discussed in Chapter 4, OpenPilot supports different safety-related functionalities, including Adaptive Cruise Control (ACC), Driver Monitoring (DM), Automated Lane Centering (ALC), Forward Collision Warning (FCW), and Lane Departure Warning (LDW). Similar to Chapter 5, for the experiments, we utilized the OpenPilot regression testing tool, Process Replay [168], and adjusted it to accept any inputs, not only predefined ones. VulFuzz++ concolic engine works at the code level of OpenPilot, not the binary level, as it is impossible to generate a binary from OpenPilot.
Table 6.1: Prioritization and branch coverage during stage two and three

<table>
<thead>
<tr>
<th></th>
<th>Queue</th>
<th>Number of Branches</th>
<th>Stage Two Test Cases</th>
<th>Stage Two Branch Coverage</th>
<th>Stage Three Test Cases</th>
<th>Stage Three Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Run High Priority</td>
<td>33</td>
<td>20</td>
<td>5,073</td>
<td>270</td>
<td>5,089</td>
<td></td>
</tr>
<tr>
<td>First Run Low Priority</td>
<td>51</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Second Run Low Priority</td>
<td>28</td>
<td>16</td>
<td>5,095</td>
<td>263</td>
<td>5,112</td>
<td></td>
</tr>
</tbody>
</table>

6.7.2 Experiments and Results

All experiments are executed on a 64-bit Ubuntu 16.04 Long Time Support (LTS) system. The machine is equipped with an Intel Core i7-1065G7 processor, a four-core chip with Hyper-Threading that runs at a base frequency of 1.3GHz, and 8GB Low-Power Double Data Rate (LPDDR4X 4267 MHZ) memory.

Evaluating VulFuzz++

We executed VulFuzz++ on OpenPilot until it halts, and analyzed the performance of each stage of VulFuzz++ in terms of the generated test cases, obtained coverage, and exposed crashes.

Test Case Analysis. In stage one of VulFuzz++, the grey-box fuzzer achieved generating 1,810 unique test cases in 16.8 hours, traversing a total of 4,812 branches. In the second stage, VulFuzz++ prioritized 84 branches that VulFuzz did not execute but belonged to traversed path functions. Table 6.1 shows the prioritization and branch coverage results. Out of the 84 branches, 33 are classified as high priority and 51 as low priority. The concolic engine attempted to satisfy the 33 branches and generated 20 test cases that increased the branch coverage to 5,073. In the third stage, the newly generated test cases are passed to the directed version of VulFuzz to be mutated, resulting in 270 test cases. Before starting the second run of stage two, the low queue is updated. Nodes that are already traversed are eliminated, and new
nodes that can increase the coverage of vulnerable functions are appended, resulting in 28 branches. The concolic engine generates 16 test cases transferred to the fuzzer in stage three, which increases the branch coverage to 5,112.

**Coverage Analysis.** Fig. 6.2 represents the statement coverage obtained by VulFuzz++, showing the progress by each stage of the framework. In the first stage, VulFuzz begins by prioritizing the testing towards vulnerable components. It thoroughly evaluates weak components in the first 15.8 hours of the testing, shown in a steady non-progressing region. VulFuzz then switches to code expansion and reaches its optimal coverage promptly presented in a sudden steep upward trajectory in the coverage plot. The grey-box fuzzer then fails to expand further the coverage, and VulFuzz++ switches to concolic execution. The results show that VulFuzz++ concolic engine achieves in supporting the grey-box fuzzer and considerably expands the code coverage. The
targeted concolic engine generates a few test cases in each round that satisfy input-specific conditions that were blocking the grey-box fuzzer from traversing deeper paths of the code. When such inputs are satisfied and the testing is lead again by the grey-box fuzzer, just a few mutations enable the fuzzer to execute new branches with large blocks of undiscovered statements, shown by a surge in statement coverage after 18.6 hours of testing. The fuzzer in stage three required only 533 test cases to expand the coverage as it focuses the testing on the nodes that were discovered in stage two instead of wasting testing time on nodes that were already traversed.

To analyze the hybrid fuzzer capabilities in satisfying input-specific conditions, we also measure the branch coverage, which provides accurate information on the transactions resolved by VulFuzz++. The results of stage two and three runs are summarized in Table 6.1. The first iteration of stages two and three increases the branch coverage by 277 branches and the second iteration by 23. Hence, VulFuzz++ boosts the grey-box fuzzing results and covers approximately 96.7% of the systems’ conditional branches. Performing more iterations of stages two and three on OpenPilot does not increase the coverage as the SMT solver already attempted to resolve the remaining branches but failed.

Crash Analysis. Increasing the coverage and assuring thorough evaluation of vulnerable components supports VulFuzz++ in the vulnerability exposure process. VulFuzz++ identified crashes are presented in Fig. 6.3, which shows the progress of different stages and runs. VulFuzz++ exposes 560 crashes, out of which 446 are unique crashes. Thus, VulFuzz++ augments the grey-box fuzzer, VulFuzz, with an additional 150 crashes, depicting a 50% increase in unique crashes. Note that the identified crashes of
OpenPilot in the simulated environment may not be applicable in real-world operations. VulFuzz++ exposed several types of vulnerabilities, including index out of bound exceptions, uninitialized values, and illegal argument exceptions. The stages of the VulFuzz++ framework work collaboratively to achieve the hybrid testing goals. Though the test cases generated by VulFuzz++ concolic engine slightly increase the crash coverage, the third stage of VulFuzz++ improves the detected crashes remarkably. When stage three leads the testing, the fuzzer extensively evaluates vulnerable functions and attempts to explore the target branches’ surrounding regions.

The sharp increase in the detected crashes presented at hour 21.2 in the plot of Fig. 6.3 shows the importance of fuzzing low priority nodes. As outlined in the framework steps, after the first run of the directed VulFuzz, if the fuzzing approaches a vulnerable function, the concolic execution attempts to resolve the branch that
enables its execution. In this experiment, VulFuzz++ adds two branches to the low priority queue. The concolic engine finds satisfying inputs for these branches and hands them over to the fuzzer in the second iteration of stage three. Between hours 21.1 and 21.2, VulFuzz++ was thoroughly evaluating the vulnerable functions. Testing the components with multiple inputs eventually led to the discovery of significantly more crashes. Hence, generating numerous test cases originating from stage two test cases allows VulFuzz++ in stage three to explore new regions, exposing several undiscovered crashes.

**Comparative Analysis Between VulFuzz++ and Other Methods**

VulFuzz++ aims to boost grey-box fuzz testing without increasing the complexity of security testing. To sum up the benefits and shortcomings of VulFuzz++, we compare the results of VulFuzz++ to the state-of-the-art fuzzer American Fuzzy Lop [107] and black-box fuzzer. As in Chapter 5, we utilized the Python fork of AFL and built an unguided black box fuzzer. To give the black-box fuzzer a higher chance of broad coverage, we designed the black-box fuzzer with the data-type mutation-based fuzzer from VulFuzz. Hence, the black-box fuzzer can generate inputs that bypass the initial validation criteria of OpenPilot but randomly attempt to validate the system. To compare the VulFuzz++ framework to approaches that support fuzzing with concolic execution, we used the concolic engine of VulFuzz++ without the prioritization and optimized input steps. The engine solely attempts to target the 84 branches that were not solved by VulFuzz and does not pass the resulting inputs back to the fuzzer again.

Existing hybrid fuzzing techniques work on the binary level of the target system.
Table 6.2: Comparison between the results of VulFuzz++ and other methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Test Cases</th>
<th>Dropped Test Cases</th>
<th>Unique Crashes</th>
<th>Branch Coverage</th>
<th>Branch Coverage (%)</th>
<th>Running Time (hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>American Fuzzy Lop (AFL)</td>
<td>1,002</td>
<td>233</td>
<td>8</td>
<td>4,809</td>
<td>90.9</td>
<td>8.7</td>
</tr>
<tr>
<td>Unguided Mutation Fuzzer</td>
<td>1,810</td>
<td>20</td>
<td>79</td>
<td>4,800</td>
<td>90.7</td>
<td>16.8</td>
</tr>
<tr>
<td>Vulnerability-Oriented Fuzzer (VulFuzz)</td>
<td>1,810</td>
<td>20</td>
<td>296</td>
<td>4,812</td>
<td>91</td>
<td>16.8</td>
</tr>
<tr>
<td>VulFuzz ∩ Targeted Concolic Execution</td>
<td>1,870</td>
<td>20</td>
<td>299</td>
<td>5,095</td>
<td>96.3</td>
<td>18.1</td>
</tr>
<tr>
<td>VulFuzz++</td>
<td>2,388</td>
<td>20</td>
<td>446</td>
<td>5,112</td>
<td>96.7</td>
<td>22.2</td>
</tr>
</tbody>
</table>

As it is not possible to generate an executable from OpenPilot, we could not compare VulFuzz++ with existing hybrid testing techniques. Regardless, Driller’s [119] continuous interweaving between fuzz testing and concolic testing slows the testing progress. The concolic engine of Driller has to look actively for branches and attempt to solve them. Thus, the engine targets are not predefined, making the execution time of the concolic engine longer than VulFuzz++, with heavy memory consumption. Moreover, QSYM [120] supports only x86_64 architecture, which is not compatible with all ECUs.

**Complementary Stages.** Table 6.2 provides an overview of the evaluation results when experimenting with different methods. VulFuzz++ generates 1,386 more test cases than AFL in 22.2 hours. AFL finishes testing considerably faster, within 8.7 hours. However, it has inadequate crash identification results. VulFuzz++ discovers 438 more unique crashes than AFL. Similarly, the unguided mutation fuzzer finalizes the testing within 16.8 hours and achieves 71 unique crashes only. Unlike AFL and common black-box fuzzers, VulFuzz is explicitly tailored to accommodate the unique automotive testing challenges. It reduces the number of dropped test cases (only 1% dropped by VulFuzz compared to 23% dropped by AFL), focusing the fuzzing testing
6.7. EVALUATION

on vulnerability exposure. Thus, VulFuzz++ is reinforced with a grey-box fuzzer that quickly and efficiently examines the automotive component reducing the reliance on concolic execution and enabling limited iterations between the two testing methods.

However, VulFuzz does not outperform the other fuzzing methods in terms of coverage, and it only accomplishes a slight improvement. Similar to other grey-box and black-box fuzzing tools, VulFuzz cannot solve input-specific conditions. VulFuzz++ concolic engine assists the grey-box fuzzer in covering 96.7% of OpenPilot branches, offering a comprehensive testing tool. Every stage of VulFuzz++ contributes toward maximizing the vulnerability exposure process. When the grey-box fuzzer VulFuzz is only supported with a concolic engine, the symbolic exploration successfully expands the coverage offering 96.3% total coverage. Nevertheless, concolic execution alone does not help in triggering new crashes. The engine discovered only three crashes. Hence, the concolic engine role is solely limited to coverage expansion, and it is inadequate for security testing that aims to expose vulnerabilities. For this reason, VulFuzz++ hands over the vulnerability identification to the fuzzer in stage three.

**Relation to Vulnerable Functions.** The core enabler for outstanding crash analysis by VulFuzz++ is the weight assignment for vulnerable components. Following the same approach presented in VulFuzz [22], VulFuzz++ uses security vulnerability metrics to calculate the vulnerability score of each function of OpenPilot. 5% of OpenPilot’s functions have a high vulnerability score and are assigned a weight of value 100. 3% of the functions have a medium vulnerability score and are assigned a weight of value 50. Functions with low vulnerability scores are not assigned any weight (value 0). The bar graph of Fig. 6.4 compares the number of unique crashes with the number of test cases that traversed a weighted function. VulFuzz++ has
Figure 6.4: The number of unique crashes and the number of times test cases execute weighted functions

the highest number of test cases that target weak components (48.6% of its test cases). On the other hand, only 7.8% of AFL test cases and 10% of unguided fuzzer test cases traverse vulnerable functions, affecting their performance in crash identification. Supporting VulFuzz with only concolic execution increases test cases that examine vulnerable functions by 2%, which is insufficient to assure a reliant automotive system. On the contrary, targeted hybrid fuzz testing that extensively validates weak components can augment vulnerability exposure. VulFuzz++ directs the testing towards weak functions, increasing the test cases that traverse weighted functions by 43%.

**Execution Time.** VulFuzz++ outperforms all other testing methods in terms of coverage and crashes identification, but it increases the testing time. As shown in Fig. 6.5, VulFuzz++ has the highest execution time of 22.2 hours. That is 13 hours
more than AFL and 5.4 hours more than the unguided fuzzer and VulFuzz. Similar to VulFuzz++, supporting grey-box fuzz testing with concolic execution increases the testing time considerably. Compared to VulFuzz++ that generates 2,388 test cases in 22 hours, VulFuzz with the concolic engine generates only 1,870 in 18.1 hours. The concolic engines spend 1.3 hours generating only 60 test cases. VulFuzz++ prioritizes the untraversed nodes and selects an optimal input to direct the concolic execution, diminishing the symbolic exploration time. However, traditional concolic engines run with several inputs attempting to solve every untraversed branch, examining the same paths multiple times, wasting hours of testing time, and causing memory overhead.

6.8 Summary

This chapter proposes a hybrid fuzz testing framework, VulFuzz++, that combines grey-box fuzzing and concolic exploration in one tool. VulFuzz++ has three stages that...
expand the coverage and boost vulnerability identification while assuring minimum overhead. The VulFuzz++ framework offers a reliable vulnerability exposure tool that promises a comprehensive and deep exploration of automotive software components. Due to strict deadlines in the automotive industry, efficiency is a fundamental factor affecting the reliability of any security testing tool.

We evaluated VulFuzz++ against an autonomous driving system, OpenPilot. VulFuzz++ successfully achieves its goals without causing significant overhead. It augments the grey-box fuzzer results and increases the discovered crashes by 50%. Moreover, it accomplishes 96.7% branch coverage. The results show that supporting grey-box fuzzing with traditional concolic execution cannot enhance crash identification; it only increases crash identification by 1%. Though VulFuzz++ increases processing time moderately, the benefits of VulFuzz++ exceed the shortcoming. For safety-critical subsystems, a comprehensive evaluation is vital to offer a reliable system.

Though security testing offers a reliable tool that recognizes vulnerabilities during the development lifecycle, it is essential to monitor the changing security risk and vulnerabilities during the entire lifespan of vehicles. The next chapter satisfies this requirement by proposing a security decay framework for CAVs.
Chapter 7

Security Decay Monitoring

Attackers’ capabilities evolve with time. What was once secure can become an easy target for skilled attackers to take advantage of systems’ weaknesses to initiate attacks. Hence, any software system should be carefully monitored to identify possible security decay that can expose it to malicious behavior. Software integration and internet connectivity expose vehicles to cybersecurity challenges that, if not handled, can lead to destructive results. It is essential to identify security decay in automobile systems.

Following security standards during Vehicle Software Engineering (VSE) helps create more resilient Connected Autonomous Vehicles (CAVs) that defend against current attacks [12]. Nevertheless, attackers’ techniques are advancing. The average age of cars and trucks is more than ten years, and future vehicles are expected to operate for even a longer period [17]. With such a long lifespan, new software vulnerabilities will be discovered, new attacker tools will be developed, and adopted security practices will become weaker. Ensuring CAV security requires planning that does not bind security assurance and risk assessment to the development phase but spans to cover the vehicles’ operation phase.
Security decay represents a drop in system resilience due to newly discovered vulnerabilities, more skilled attackers, or changes in the operating environment of vehicle software.

This research aims to identify security decay across vehicle software systems’ full lifespan. We achieve this by identifying vulnerabilities in the system and assessing the evolving risks. We propose an Autonomous Vehicle Security Decay Assessment (AVSDA) framework composed of two phases. Considering the size and complexity of vehicle software systems, security risk estimation of all components becomes unmanageable. The first phase, vulnerability analysis, automatically and efficiently identifies potentially weak or vulnerable components. The second phase, risk analysis, focuses on quantifying the risk of weak components by determining an attack’s likelihood and assessing its impact.

Traditional threat and risk assessment methods (e.g., E-Safety Vehicle Intrusion Protected Applications (EVITA) threat and risk model [18]) determine security risks by identifying and classifying potential threats. In contrast, we identify security risks by targeting the source of issues. The AVSDA framework distinguishes the vulnerable components that are responsible for the vast majority of vehicle cyberattacks. Considering the operational environment of vehicles, quantifying various threat scenarios becomes a daunting task. Hence, vulnerability analysis is used to efficiently measure the weak components that make vehicle software systems unprotected against attacks (e.g., unauthorized access to data, acceptance of bogus information, and unauthorized control of vehicles).
This chapter outlines the AVSDA framework and discusses its phases while highlighting the importance of assessing autonomous system security decay at the software level to help prevent malicious behavior and maintain vehicle safety. We end this chapter by presenting the results from applying the AVSDA framework to OpenPilot.

7.1 Framework Design

This section introduces the Autonomous Vehicle Security Decay Assessment (AVSDA) framework and discusses its phases. We begin by providing an overview of AVSDA and then dive deeper into the framework phases.

The AVSDA framework aims to identify security decay of autonomous vehicle software systems. The proposed framework analyzes the security decay at the Software Component (SWC) level defined in Chapter 4 as a structural element that provides an interface. It can utilize different automotive communication means and is connected to other parts to fulfill a function.

Unauthorized access, acceptance of falsified information, interruption of service, and unauthorized control are all types of threats that an attacker can initiate to jeopardize vehicle software systems. Such threats violate vehicles’ security and can threaten individuals’ safety. The AVSDA framework utilizes a vulnerability analysis approach to estimate the vehicle software system’s weak components that facilitate the existence of security threats.

As depicted in Fig. 7.1, AVSDA comprises two phases: vulnerability analysis and risk analysis. The first phase identifies potentially vulnerable components that can cause security failures. Attackers take advantage of existing software defects to initiate malicious behavior. Hence, to detect security decay, we first quantitatively identify
the weak components based on the security metrics designed to target autonomous systems’ vulnerabilities. The second phase thoroughly analyzes the system’s vulnerable components to determine an attack’s likelihood and impact. It also identifies security risk level for vulnerable components.

Assessing security risks before a product release is essential to prevent catastrophic results. AVSDA should be applied before moving a vehicle model to production and periodically during the operation phase to help security engineers measure vulnerabilities, estimate changing risk levels, and avoid unwanted security breaches. Comparing
the results of subsequent runs can help security engineers in identifying system security decay. An increase in the security risk level can alert security engineers to apply the proper mitigation measures.

### 7.1.1 Vulnerability Analysis Phase

Vulnerabilities in autonomous systems are common, and the problem is growing as we move towards full automation with vehicles that entirely run on code without drivers’ intervention [173]. Vulnerabilities are usually introduced to a system during the development phase. However, they remain in systems for many reasons, including insufficient security testing, lack of security experience, weak security practices, and lack of planning. Vehicle Software Engineering (VSE) faces many challenges, including a lack of expertise and software testing complications [12]. These challenges can increase the likelihood of vulnerabilities. Hence, it is essential to identify weaknesses before they are exposed and lead to successful attacks.

The first phase of AVSDA, vulnerability analysis, measures the security vulnerability score of every component using the security vulnerability metrics introduced in Chapter 4. Components with a high security vulnerability score are further analyzed in the risk analysis phase. Measuring security vulnerability involves six steps: (1) Compute code complexity, (2) Measure component coupling, (3) Identify input and output data vulnerability, (4) Discover past security issues, (5) Compute component maturity, and (6) Calculate security vulnerability score. Steps 1 through 5 can run simultaneously, and the results of these steps are utilized in the final assessment (Step 6).

In this phase, we consider the unique architecture of vehicles and the specific
development challenges of vehicle software systems. Since vehicle software systems have vulnerability factors similar to other software systems, some of the steps of this phase can be applied to other systems (Steps 1, 4, and 5). Steps 2 and 3 are specific to autonomous vehicles. Step 2 measures how reliant a component is on other subsystems by defining the set of reachable Electronic Control Units (ECUs) from a component ECU. Step 3 identifies input and output data risks. We consider the different communication means that transmit inputs and outputs within vehicle software systems and the various threat levels that each poses on a vehicle.

7.1.2 Risk Analysis Phase

The vulnerability analysis phase and risk analysis complement each other in identifying system security decay. The second phase of the decay model examines the potentially weak entities closely and quantifies the system’s overall risk level. The risk analysis phase involves five steps: (1) Measure attack surface, (2) Estimate attacker threat, (3) Estimate likelihood of an attack, (4) Estimate impact level, (5) and Identify security risk.

The risk analysis phase is tailored to accommodate the uniqueness of vehicle software systems. For example, Step 1 of the second phase starts by measuring vehicle software systems’ attack surfaces. We consider all communication means used by vehicle software systems. Moreover, we describe the attacker threat and impact level parameters specifically for vehicle software systems.
Measure Attack Surface

The attack surface of a component is the subset of system resources that an attacker can use to initiate malicious behavior \[174\]. To conduct an attack, malicious users connect to one of the vehicle’s networks and invoke certain functions to send or/and receive information. For example, in 2015, a simulated attack was initiated on Jeep Cherokee while operating on the highway. Using the telematics system’s Wi-Fi connection, the attackers transmitted messages to disable the brakes and halt the engine functions \[175\]. Hence, an attacker usually connects to one of the system’s channels, invokes some methods, and sends data items to establish an attack on the system.

The attack surface examines the sets of entry points, exit points, communication channels, and untrusted data of a system \[176\]. The entry point set holds the means through which data can enter into the autonomous software system from the vehicle’s environment (e.g., user inputs, sensor inputs, and incoming signals). The exit point set carries the means that enable data to exit from the system (e.g., outgoing signals). There exist various communication channels that an attacker can use to connect to a vehicle. A remote attack in the vehicle software system may occur through long-distance communication mechanisms such as cellular and satellite radio. Access to the vehicle’s on-board diagnostics (OBD) port permits physical attacks that enable attackers to connect to the internal vehicle network \[88\]. In between are close-range wireless communications such as Near Field Communication (NFC) and Bluetooth, which can be utilized to perform remote attacks with nearby relays and proxies. Finally, the untrusted data set contains persistent data items stored on the nonvolatile memory of ECUs to send or receive data indirectly.
The input and output data vulnerability (DV) metric calculates the risk of inputs and outputs considering their type and the used mean of communication. Nevertheless, not all inputs and outputs can be used in an attack. The attack surface metric includes only the resources contributing to an attack. Hence, in this step, some manual validation is required. We closely look at the DV metric result to identify which elements can ease attacks.

The attack surface (AS) of each component is assigned one of the three levels\(^1\): large (value of 8), medium (value of 3), and small (value of 1). Large AS indicates that the component’s attack surface sets expose the system to multiple attacks. Medium AS means that the attack surface of the component indicates the possibility of some attacks. Small AS suggests a very low probability of initiating an attack on this component. The AS of component \(C\) is then estimated using Equation \(7.1\), where \(\omega_a\) is a weight\(^2\) assigned by security experts to emphasize the importance of the attack surface, and \(L_a\) is the value assigned based on the level.

\[
\text{AS}(C) = \omega_a L_a
\]  
\(7.1\)

\(^1\)We define specific level values to rate the risk. These values are identified to reflect the level of risk and enable quantitative measurement. Different risk values have comparable ranges to reflect various risk levels accurately. Consistently, the highest risk level between different parameters has a value of 8, and the lowest has 1. In between these two levels, values are assigned depending on the number of medium levels (e.g., one medium level assigned value 3, two medium levels assigned values 4 and 2). Security engineers can assign other values but have to follow the same approach assuring proportional ranges in the risk values of different levels.

\(^2\)Weights are used to assign the degree of importance to the attributes. Values assigned to the weights should fall within a predefined range and be used consistently for the same degree of importance. For example, the highest importance weight can have a value of 8, medium importance can have a value of 4, and low importance can have a value of 1.
### 7.1. FRAMEWORK DESIGN

Table 7.1: Attacker threat parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Level</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skill</td>
<td>Technical experience an agent should possess.</td>
<td>Non-specialists: No experience is required to conduct an attack.</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Skilled: Some experience is expected in the fundamentals of technology to initiate a successful attack.</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specialist: Profound knowledge in attacking techniques is required to break the system.</td>
<td>1</td>
</tr>
<tr>
<td>Knowledge</td>
<td>Background knowledge an agent should acquire about the vehicle software system architecture.</td>
<td>Public: Information needed to attack a system is publicly available (e.g., standards and protocols)</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Restricted: Data required to initiate an attack is shared with partners and protected by non-disclosure agreement.</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Private: Sensitive data shared internally with specific members is needed to conduct an attack.</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Critical: Information required to conduct an attack is strictly shared with few members (e.g., cryptography keys).</td>
<td>1</td>
</tr>
<tr>
<td>Equipment</td>
<td>Software tools and hardware tools needed to attack a vehicle.</td>
<td>Standard: Tools needed are cheap and broadly available (e.g., RTL-SDR).</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sophisticated: Obtaining the equipment is not easy and expensive.</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rare: Equipment required is not available and may entail designing or producing a sophisticated tool.</td>
<td>1</td>
</tr>
<tr>
<td>Opportunity</td>
<td>The time and attack type (remote, physical) needed to break the system.</td>
<td>Large: Attacking the system takes a short time and can be conducted remotely.</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medium: Attacking the system needs some time, and either physical or remote access is required.</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Small: Attacking the system requires much time with physical and remote access to achieve the attack.</td>
<td>1</td>
</tr>
</tbody>
</table>

Estimate Attacker Threat

We take a closer look at the agent that initiates a threat. This step of the framework is vitally important to identify security decay. As discussed earlier, attackers’ experience and knowledge are always evolving, which affect the security of the system and make it weaker. To estimate the attacker threat (AT), we employ four parameters: Skill, Knowledge, Equipment, and Opportunity. Table 7.1 describes the parameters and assigns values based on the defined levels. Similar parameters are utilized in the literature with slightly different definitions \[105, 177, 178\]. The AT of component \( C \) is estimated using Equation 7.2, where \( P \) is the set of parameters, \( \omega_p \) is the weight, and \( T_p \) is the value of the parameter.
Estimate Likelihood of an Attack

An attack takes place by an agent that targets vehicle system vulnerabilities. Hence, to estimate the likelihood of an attack (LA) Equation 7.3 is used, multiplying the attack surface probability with the attacker threat probability.

\[ LA(C) = AS(C) \times AT(C) \]  
(7.3)

Estimate Impact Level

Attacks’ impact may vary significantly; some may cause minor issues that do not necessitate a rapid response, while others can have devastating outcomes that require prompt resolution. We estimate the impact level (IL) with five parameters: Safety, Operational, Financial, Privacy, and Reputational. Vehicle attacks can affect different parties in the vehicle industry, including passengers, drivers, pedestrians, vehicle manufacturers, and associated companies. The five defined parameters estimate the impact considering all these parties. For example, the safety parameter evaluates the direct physical damage caused by an attack on the vehicle users, while the reputational parameter considers the indirect harm to manufacturers. The parameters are presented in detail in Table 7.2 with values\(^1\) based on the impact level. We define the safety impact levels based on ISO 26262 [11], a well-established safety standard for vehicles. Similar parameters are utilized in the literature, but we tailor the parameters’ level to fit the vehicle industry [4,18,179,180]. To estimate the IL of an attack

\[ AT(C) = \sum_{p=1}^{P} \omega_p T_p \]  
(7.2)
Table 7.2: Impact level parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Level</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety</td>
<td>Safety of vehicle passengers, pedestrians, and road users.</td>
<td>High: Life-threatening injuries with the possibility of casualties.</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medium: Critical injuries with the possibility of survivals.</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low: Moderate injuries with the assurance of survivals.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>None: No injuries.</td>
<td>0</td>
</tr>
<tr>
<td>Operational</td>
<td>Interruption of vehicular services.</td>
<td>High: Loss of significant subsystem in the vehicle that causes</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medium: Some functionalities within the vehicle system may not be</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>operating correctly without affecting passengers’ safety and</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driving conditions.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low: Minor operations are interrupted that do not affect the</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vehicle performance (e.g., audio services, calling services).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>None: No interruption</td>
<td>0</td>
</tr>
<tr>
<td>Financial</td>
<td>Direct and indirect financial losses affecting the vehicle owner and</td>
<td>High: Enormous financial damages that leave the vehicle</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>manufacturer.</td>
<td>manufacturer with bankruptcy risk.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medium: Significant financial losses that slightly affect the</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>financial situation of the manufacturer.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low: Minor financial losses that do not affect the</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>manufacturer operation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>None: No losses.</td>
<td>0</td>
</tr>
<tr>
<td>Privacy</td>
<td>Damages caused by data misusage, including users and manufacturer</td>
<td>High: Data leakage and privacy violations affecting</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>information.</td>
<td>a high number of users.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medium: Data leakage and privacy violations affecting</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a small number of users.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low: Minor privacy violation without any data leakages.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>None: No data misusage.</td>
<td>0</td>
</tr>
<tr>
<td>Reputational</td>
<td>Damages that affect the reputation of the manufacturer organization.</td>
<td>High: Loss of a large number of customers and shareholders with the</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inability to recover and restore a good reputation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medium: Loss of some customers.</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low: Some unsatisfied customers that can be compensated.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>None: No damages.</td>
<td>0</td>
</tr>
</tbody>
</table>

on Component $C$, Equation 7.4 is used. $F$ is the set of attack impact parameters, $\omega_f$ is the weight\(^2\) of the parameter assigned by security specialists, and $I_f$ is the value of this parameter.

$$IL(C) = \sum_{f=1}^{F} \omega_f I_f$$  \hspace{1cm} (7.4)

**Identify Security Risk**

The final security risk (SR) of a component is obtained using Equation 7.5, which links the likelihood of an attack with the impact level. The security measures applied
to protect the vehicle can lessen the threat. Thus, when estimating the SR, security specialists have to analyze and review the security controls adopted within a component to assess their ability to protect the attack surface and diminish attackers’ capabilities. According to the analysis, a vehicle component’s SR is classified into three levels: low, moderate, and severe. Low level means that the component is not under risk. This could be due to the security measures applied or because an attack probability is very low. Moderate level indicates that an attack risk exists. However, countermeasures can be used to lessen the risk. Severe level indicates that the component is facing very high risk, and the result of an attack may be critical. Applying security measures at this level might not be sufficient.

\[ SR(C) = LA(C) \times IL(C) \]  

(7.5)

7.2 Case Study

This section demonstrates the use of the AVSDA framework. Similar to the preceding chapters, we utilize OpenPilot (Version 0.7.8) [156] in our case study. OpenPilot has one component only, Autopilot. We apply the AVSDA phases on the Autopilot component, illustrating the usefulness of this framework. We finalize this section by demonstrating the importance of applying AVSDA periodically.

7.2.1 Vulnerability Analysis

We show that the vulnerability analysis phase of AVSDA can be automated efficiently to identify potentially weak components. We apply five steps of the vulnerability analysis phase to OpenPilot, and the results are detailed in Chapter 4 and
7.2. CASE STUDY

### Table 7.3: Vulnerability analysis of OpenPilot

<table>
<thead>
<tr>
<th>Steps</th>
<th>Value</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Code Complexity (CX)</td>
<td>$CX(OpenPilot) = 52,608 + 2(6,298) + 3(6,148) = 83,648$</td>
<td>The system has a total of 52,608 SLOC, 6,298 ND, and 6,148 NOC. Since ND and NOC are associated with vulnerabilities, we give them weights of 2 and 3, respectively.</td>
</tr>
<tr>
<td>Measure Component Coupling (CC)</td>
<td>$CC(OpenPilot) = 9$</td>
<td>OpenPilot communicates with the Engine Control Module, Brake Control Module, Safety System, Seat Control Unit, Powertrain Control Module, Transmission Control, Telematics Control Unit, Active Front Steering, and Battery Junction Box.</td>
</tr>
<tr>
<td>Identify Input and Output Data Vulnerability (DV)</td>
<td>$DV(OpenPilot) = 242 + 2(407) + 3(397) + 5(1) + 5(1) + 15 + 2(73) + 5(211) + 5(1) = 3,478$</td>
<td>OpenPilot’s defined inputs and outputs are all fluctuating. The system receives and sends data using serial communications. (inputs: 242 and outputs: 15), Controller Area Network (CAN) (inputs: 407 and outputs: 73), Global Positioning System (GPS) (inputs: 397 and outputs: 0), Vehicle to Infrastructure (V2I) (inputs: 1 and outputs: 211), and User to Vehicle (U2V) (inputs: 1 and outputs: 1) communications. We assign different weights for these communication means as they pose different risks.</td>
</tr>
<tr>
<td>Discover Past Security Issues (PSI)</td>
<td>$PSI(OpenPilot) = (0.5)^2 + 18(0.5^3) + 50(0.5^4) = 50.25$</td>
<td>There are 69 reported bugs in OpenPilot reported since 2018 (1 in 2018, 18 in 2019, and 50 in 2020). Higher weight is assigned to attacks that occurred in 2020.</td>
</tr>
<tr>
<td>Compute Component Maturity (CM)</td>
<td>$CM(OpenPilot) = 100 \left( \frac{30,688}{52,608} \right) = 58$</td>
<td>Within OpenPilot, 30,688 SLOC is modified. None of the applied changes are labeled as security enhancement or maintenance.</td>
</tr>
</tbody>
</table>

summarized in Table 7.3.

#### 7.2.2 Risk Analysis

To validate the risk analysis phase’s applicability, we apply the steps to OpenPilot and show their usefulness. We measure the attack surface (AS) of OpenPilot by reviewing the system’s inputs, outputs, channels, and methods. All the system inputs are fluctuating and OpenPilot utilizes multiple communication means, increasing the attack surface. For example, users can connect their smartphones to OpenPilot, exposing the vehicle to different remote attacks [88]. OpenPilot uses nonvolatile memory, allowing untrusted data to be stored. Accordingly, AS’s level is considered as large (value of 8), and we assign a weight of 3 to emphasize the criticality of the attack surface. According to Equation 7.1, the estimated value of AS is 24.
Next, we estimate attacker threat (AT) using the parameters of Table 7.1. Attacking the Autopilot system requires outstanding experience in different domains, including networking and security. Hence, specialist skill (value of 1) is required to pose a risk on the system. Some background knowledge about the system is required to initiate an attack. Since OpenPilot is open source, we assign the knowledge parameter a value of 8. Moreover, the equipment needed to conduct an attack is standard (value of 8), like a computer and ports. Finally, performing an attack on the Autopilot system requires preplanning, and either physical or remote access is needed (value of 3). With these level values, AT is 20. The likelihood of an attack (LA) can now be estimated 480 based on Equation 7.3.

Next, we determine the impact level of an attack (IL). Establishing an attack on an autopilot system might not have severe direct consequences but can lead to drastic indirect results. The vehicle can operate without autopilot functionality. However, such functionality communicates with critical components (e.g., engine and brake ECUs). If a malicious attack successfully propagates, the vehicle’s safety and operational status are left in critical condition. Hence, the safety and operational parameters are both at a high level (value of 8). The manufacturing company might face significant financial losses (value of 3) when all the models affected by such an attack are recalled. Moreover, OpenPilot collects data, including locations, Controller Area Network (CAN) messages, and road conditions. The leakage of such data can violate the privacy of affected users only (value of 3). Finally, such attacks have a moderate reputational impact, with some possible customer loss (value of 3). After estimating all the parameters of Table 7.2, the impact level (IL) can be determined using Equation 7.4. We assign a weight of 4 for the safety and operational parameters to
emphasize their importance, and the final value of IL is 73.

The last step is determining the security risk (SR) based on the likelihood of an attack, impact level, and practiced security measures. First, we evaluate SR using Equation 7.5, which results in 35,040. Then we review OpenPilot’s security practices to identify the SR level. OpenPilot follows MISRA c2012 [34] software development guidelines, preventing common coding errors. However, this is not enough to mitigate all security issues. Accordingly, we assign a moderate SR level, which indicates that an attack risk exists.

7.2.3 Framework Application Frequency

The AVSDA framework should be applied periodically to identify security decay. For example, consider a vehicle attack technique becomes publicly available with a video explaining how to accomplish the attack. The effect of such an incident on the vehicle system’s security can be detected by the AVSDA framework. Attacker threat skill parameter is changed from a specialist to a non-specialist with a value of 8. Accordingly, AT rises to 27, increasing the likelihood of an attack to 648. The system’s security risk extends from 35,040 to 47,304, indicating a security decay and the need for applying robust security measures to defend the vehicle.

7.3 Summary

We propose an Autonomous Vehicle Security Decay Assessment (AVSDA) framework that estimates the security decay of vehicle software systems by quantitatively measuring systems’ vulnerabilities and risks. The AVSDA framework is composed of
two phases. The vulnerability analysis phase uses security metrics to identify vulnerable components. The risk analysis phase carefully evaluates attack likelihood by identifying the attack surface and estimating attackers’ threats. The framework further analyzes attacks’ severity by assessing their impact. The final step of the risk analysis phase defines security risk based on the applied security measures. The AVSDA framework should be applied periodically to recognize changes in the security risk and possible decay.

We evaluated AVSDA vulnerability analysis phase metrics’ performance by experimenting with their usefulness in identifying vulnerabilities of OpenPilot, an Autopilot system. The results show that the framework is capable of identifying vulnerabilities with an accuracy rate of 94%. The case study shows the efficiency of AVSDA in systematically estimating security risks and discovering security decay.
Chapter 8

Conclusions, Limitations, and Future Work

Securing the vehicle requires adopting different policies and techniques to secure the vehicle’s network, hardware, and software systems. Protecting all these components is essential to build a safe, reliable, and trustworthy vehicle. Many successful attacks can be related to software components. Improper design and specification cause several vulnerabilities that expose the system to different threats \cite{145, 181}. For example, many researchers linked existing attacks to weak input validation by automotive software systems \cite{58, 182}. Hence, granting resilient CAVs is an unattainable goal without embedding security consideration into all the development phases of automotive systems.

It is vital to verify the security of automotive systems by detecting software defects and security loopholes before production. Adequate security verification requires employing security testing tools that manage the security challenges of automotive systems and identify vulnerabilities, diminishing the risk and making CAVs safer. Sustaining the resilience of automotive systems requires continuous monitoring of security. As vehicles operate on the road for an extended period, embedded security
practices might debilitate against more advanced threats. Thus, the automotive industry needs an efficient solution to recognize security decay and maintain the primary goal of CAVs, offering a safe and reliable driving experience.

8.1 Conclusions

This thesis strives to make the security journey of automotive software systems manageable and effective. We achieve our goal by designing and developing novel practical security solutions that address the unique architecture of CAVs while assuring security.

The uniqueness of CAVs originates challenges for Vehicle Software Engineering (VSE) that render traditional models and practical solutions for software development ineffective and inapplicable. This thesis reviews current practical software solutions, including standards, tools, languages, and research efforts to understand the evolution, trends, and current practice in this research area. Moreover, we offer a thorough review of existing software engineering processes analyzing their benefits and shortcomings in the context of CAVs. The analysis shows that V-Model can assist in handling the challenges and vehicles architecture better than the Agile model. This survey enables automakers and software providers to properly differentiate between existing software engineering processes and current practical software development solutions. Hence, we guide the automotive industry towards the most suitable VSE model and practical solutions that can adequately manage their challenging needs.

Producing a reliable vehicle requires incorporating security consideration in all the phases of the Software Development lifecycle (SDLC). This thesis introduces the Secure Vehicle Software Engineering (SVSE) lifecycle that runs with the software
and safety development processes assuring security-by-design. The SVSE lifecycle comprises four stages: planning, development, testing, and operation with continuous cybersecurity decay review. The proposed lifecycle incorporates several security activities that manage the challenges of security development and assure compliance with international security standards. Thus, the SVSE lifecycle promises deliverability and controllability of software security for automotive systems.

Taking into account the complexity of automotive systems, prioritization the testing is vital to ensure efficient testing. This thesis introduces five security vulnerability metrics (1) Code Complexity, (2) Component Coupling, (3) Input and Output Data Vulnerability, (4) Past Security Issues, and (5) Component Maturity. Collaboratively, these metrics identify the weak components of the automotive system that pose a high risk. We implemented and applied the proposed security metrics on an automotive system, OpenPilot, that offers an Autopilot functionality to various car models, including Honda, Toyota, Hyundai, Nissan, and Kia. The assessment shows that the metrics can distinguish vulnerable components with an accuracy rate of 94%, outperforming existing security vulnerability metrics that are not designed for automotive systems. Therefore, these metrics pave the road for dynamic security testing that prioritizes the weakest components of the system.

One of the most robust and efficient security testing methods is fuzzing. Though black-box fuzzing can validate the system with various scenarios, its blindness prevents it from exploring the deep paths of the system. We address the limitations of black-box fuzzing by introducing an efficient Vulnerability-oriented fuzz testing framework (VulFuzz) that is more knowledgeable about the system. VulFuzz has four engines (1) Vulnerability, (2) Mutation, (3) Evaluation, and (4) Prioritization.
that cooperate to prioritize the testing towards the most vulnerable components. The
grey-box fuzzer utilizes the security vulnerability metrics presented in this thesis to
identify weak components and monitors the test cases’ traversal to direct the testing
towards these components, assuring a thorough examination of weak components. In
addition, to reduce the number of dropped test cases that do not validate automo-
tive systems, VulFuzz introduces a mutation engine that infers the inputs’ data types
and performs a single mutation while preserving the fields’ data type. We experi-
mented with VulFuzz on OpenPilot, and the grey-box fuzzer achieved in identifying
296 unique crashes 3600% more than the state-of-art fuzzer American Fuzzy Lop
(AFL) and 275% more than a black-box unguided mutation fuzzer. Thus, VulFuzz
offers a robust security testing framework that can be used during the cybersecurity
component testing phase of the SVSE lifecycle to examine automotive components
with various inputs effectively.

Though grey-box fuzzing is an efficient and potent security testing tool, the fuzzer
can fail to satisfy input-specific conditions, hindering the testing from exploring
deeper paths of the automotive system. We overcome this limitation by designing
a hybrid fuzzing framework for vehicles (VulFuzz++) that utilize a targeted concolic
engine to satisfy input-specific conditions, assuring a comprehensive evaluation of the
system. VulFuzz++ offloads most of the exploration process to VulFuzz. When the
fuzzer halts failing to explore different paths, VulFuzz++ examines the untraversed
branches and prioritizes them based on their potential to expose vulnerabilities. It
utilizes a tailored, targeted concolic engine that limits the symbolic exploration to
only specific functions. When the concolic engine discovers new system inputs, test-
ing is handed over again to the fuzzer to perform a quick and efficient evaluation of the
newly explored region. We implemented and experimented with the VulFuzz++ framework on OpenPilot. VulFuzz++ boosted the vulnerability exposure process of grey-box fuzzing, increasing the obtained crashes by 50%. It dramatically outperforms traditional concolic engines in assisting fuzzers, exposing 50 times more unique crashes. VulFuzz++ assures a comprehensive examination covering 96.7% of automotive system branches. Hence, the hybrid fuzz testing framework expands the vulnerability exposure during the testing stage of the SVSE lifecycle by evaluating the deep and vulnerable paths of automotive components.

This thesis proposes the Autonomous Vehicle Security Decay Assessment (AVSDA) framework composed of two phases vulnerability analysis and risk analysis that together identifies a change in the security risk of automotive systems. The vulnerability analysis phase of the AVSDA framework also utilizes the security vulnerability metrics of this thesis to automatically and efficiently identify potentially weak components. The analysis phase further examines the weak components and quantitatively identifies their security risk, considering the attack’s likelihood and impact. Accordingly, the AVSDA warns security specialists about severe security decay that might require an immediate update or even vehicle recalls to prevent incidents. This framework is critical for the United Nations Economic Commission for Europe (UNECE) WP.29 cybersecurity compliance [138].

8.2 Limitations

In general, the main limitation in this works lies behind the unavailability of open-source automotive systems. We have evaluated the security vulnerability metrics on OpenPilot, an autonomous driving system. However, a large data set would have
proven the effectiveness of our metrics thoroughly. We could not apply the metrics on different systems to study which parameters reflect the vulnerabilities of automotive systems better. Similarly, for VulFuzz, we experimented with the framework on different processes of OpenPilot and compared the performance of VulFuzz with traditional fuzz testing approaches. However, we could not apply VulFuzz to different automotive components. Moreover, as OpenPilot is built using both Python and C languages, we could not compare VulFuzz and VulFuzz++ with some existing fuzz testing tools (e.g., AFL++ [183], and Driller [119]) as they do not support systems built in the Python languages.

The AVSDA framework can recognize security decay when applied to the same system several times over an extended period. However, OpenPilot is relatively new, and its vulnerability data is not well documented. Hence, we could not investigate the AVSDA framework over a long time.

The practical security solutions presented in this thesis aim to address the challenges of cybersecurity assurance in the automotive industry and offer reliable and efficient solutions to mitigate vulnerabilities. The security vulnerability metrics, VulFuzz, and VulFuzz++ manage the system complexity and size, input and output fluctuation, and the outsourcing challenges. However, they work at the software level and do not validate the system with the same conditions as a real-world scenario. They instead simulate sensor inputs. As discussed in this thesis, security assurance in the automotive industry require employing different testing methods to manage all the challenges and adequately validate the system.
8.3 Future Work

This section discusses possible future directions that address the above listed limitations and expand our research further.

Detecting the evolving risk throughout the entire lifespan of vehicles is essential. Hence, we intend to experiment with the AVSDA framework further in the future once more data becomes available.

Moreover, we aim to proceed in exploring the capabilities of VulFuzz and VulFuzz++ in identifying vulnerabilities. We will investigate the frameworks with different Internet of Things (IoT) systems. Heterogeneous IoT systems have different challenges than automotive systems. This expansion requires redesigning some VulFuzz engines, particularly the vulnerability engine that identifies weaknesses based on CAVs architecture. Nevertheless, the fundamental concept behind VulFuzz and VulFuzz++ frameworks can be a potent security verification tool for any large and complex system that depends on different communication means.

As the automotive industry moves toward full automation, the reliance on software increases even more. This, in return, is pushing the automotive industry towards outsourcing more. However, managing the complex supply chain of the automotive systems makes security assurance harder. We plan to improve the security vulnerability metrics to practically measure weaknesses at the binary level of a component without requiring the source code. This approach can aid the automotive industry in evaluating outsourced components quantitatively and efficiently.

Finally, the security assurance methods presented in this thesis identify vulnerabilities in the automotive systems, but they do not validate the resilience of CAVs.
For example, detecting poor authentication and defense mechanisms cannot be re-
vealed with fuzz testing. On the contrary, penetration testing exploits vulnerabilities
in automotive systems and provides an accurate depiction of CAVs resilience. How-
ever, it is time-consuming and requires rare expertise. Therefore, we plan to expand
our research in this direction by enhancing and automating penetration testing in the
automotive industry.


[119] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshi-
taishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through se-

[120] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “QSYM: A practical concolic exe-
cution engine tailored for hybrid fuzzing,” in 27th USENIX Security Symposium


gov/Projects/ssdf.

software-security/secure-sdlc.

tive development,” in 2016 23rd Asia-Pacific Software Engineering Conference

automotive industry,” International Journal of Production Economics, vol. 91,


