
by

Rahul Kumar Dutta

LIU-IDA/LITH-EX-A--15/054—SE

2015-09-18
Final Thesis


by

Rahul Kumar Dutta

LIU-IDA/LITH-EX-A--15/054—SE

2015-09-18

Supervisor: Ke Jiang
Examiner: Dr. Unmesh D. Bordoloi
Abstract

Security in automotive industry is a thought of concern these days. As more smart electronic devices are getting connected to each other, the dependency on these devices are urging us to connect them with moving objects such as cars, buses, trucks etc. As such, safety and security issues related to automotive objects are becoming more relevant in the realm of internet connected devices and objects. In this thesis, we emphasize on certain factors that introduces security vulnerabilities in the implementation phase of Software Development Life Cycle (SDLC). Input invalidation is one of them that we address in our work. We implement a security evaluation framework that allows us to improve security in automotive software by identifying and removing software security vulnerabilities that arise due to input invalidation reasons during SDLC. We propose to use this framework in the implementation and testing phase so that the critical deficiencies of software in security by design issues could be easily addressed and mitigated.

Keywords: Security testing, ISO26262, fuzzing, static analysis, error propagation, vulnerability analysis
Acknowledgements

I would like to thank Dr. Mafijul Islam and Christian Sandberg at Volvo Group Trucks Technology for being my supervisors and helping me with their valuable thoughts, guidance and support in this project. I would also like to thank Dr. Unmesh D. Bordoloi for being my mentor, supervisor and examiner and also thanks to Ke Jiang at Linköping University for providing useful inputs being my academic supervisor.

This thesis was partly funded by the HEAVENS consortium partners and it was carried out at the Advanced Technology and Research Centre in Volvo Group Trucks Technology at Gothenburg.
Contents

Abstract ................................................................................................................................. i
Acknowledgements .............................................................................................................. iii
Abbreviations ....................................................................................................................... vi
List of Figures ...................................................................................................................... vii
List of Tables ....................................................................................................................... viii

1 Introduction .......................................................................................................................... 1
  1.1 Research Questions ........................................................................................................ 3
  1.2 Project Specific Goals ..................................................................................................... 4
  1.3 Research Methodology .................................................................................................. 4
  1.4 Stakeholders .................................................................................................................... 5
  1.5 Outline ............................................................................................................................ 6

2 Security Concepts and Related Work .................................................................................. 7
  2.1 Security Testing .............................................................................................................. 8
  2.2 Software Testing methods and Tools ............................................................................. 10
  2.3 Secure Software Development Lifecycle ....................................................................... 14
  2.4 Guidelines to Safe Coding ............................................................................................ 17
  2.5 Vulnerability Analysis .................................................................................................... 20
     2.5.1 Default Pattern List ................................................................................................. 20
     2.5.2 Preliminary List of Vulnerability Examples for Researchers (PLOVER) .................. 20
     2.5.3 Seven Pernicious Kingdom ..................................................................................... 21
  2.6 Threat Analysis ............................................................................................................... 22
  2.7 Related Work .................................................................................................................. 22
     2.7.1 External Attacks on automotive software ................................................................. 23
     2.7.2 Internal Attacks on automotive software ................................................................. 24
     2.7.3 Other External and Internal Attacks ....................................................................... 24
     2.7.4 Security Framework ............................................................................................... 25

3 Automotive Concepts ......................................................................................................... 28
  3.1 Automotive Electrical and Electronic (E/E) System Architecture .................................. 29
  3.2 Automotive Software ..................................................................................................... 30
  3.3 Functional Safety Standard ............................................................................................ 31
  3.4 Automotive Safety and Security .................................................................................... 32
     3.4.1 Interplay between Safety and Security ................................................................. 32
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASLR</td>
<td>Address Space Layout Randomization</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>Automotive Open System Architecture</td>
</tr>
<tr>
<td>BSW</td>
<td>Basic Software</td>
</tr>
<tr>
<td>BUFF</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CCU</td>
<td>Capture Control Unit</td>
</tr>
<tr>
<td>CDD</td>
<td>Complex Device Driver</td>
</tr>
<tr>
<td>CVE</td>
<td>Common Vulnerabilities and Exposures</td>
</tr>
<tr>
<td>CWE</td>
<td>Common Weakness Enumeration</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>E/E</td>
<td>Electrical and Electronic</td>
</tr>
<tr>
<td>HEAVENS</td>
<td>HEAling Vulnerabilities to Enhance Software Security and Safety</td>
</tr>
<tr>
<td>MCAL</td>
<td>Microcontroller Abstraction Layer</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller Unit</td>
</tr>
<tr>
<td>MISRA</td>
<td>Motor Industry Software Reliability Association</td>
</tr>
<tr>
<td>NVD</td>
<td>National Vulnerability Database</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OWASP</td>
<td>Open Web Application Security Project</td>
</tr>
<tr>
<td>PLOVER</td>
<td>Preliminary List of Vulnerabilities for Researchers</td>
</tr>
<tr>
<td>RSU</td>
<td>Road Side Unit</td>
</tr>
<tr>
<td>RTE</td>
<td>Runtime Environment</td>
</tr>
<tr>
<td>SDL</td>
<td>Security Development Lifecycle</td>
</tr>
<tr>
<td>SDLC</td>
<td>Software Development Lifecycle</td>
</tr>
<tr>
<td>STRIDE</td>
<td>Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>SWC</td>
<td>Software Component</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1.1: Needs for cyber vehicle security ............................................................. 1
Figure 1.2: Research methodology adapted from K. Peffers .................................. 4
Figure 2.1: Total CVE flaws published on NVD from 2005 until 2015 ................... 8
Figure 2.2: Testing for what it is NOT supposed to do ........................................... 9
Figure 2.3: Securing applications – Three high level strategies ......................... 14
Figure 2.4: The Microsoft Security Development Lifecycle – Simplified ........ 16
Figure 2.5: Types of vulnerabilities from 1999-2015 ....................................... 17
Figure 3.1: E/E automotive systems architecture .............................................. 29
Figure 3.2: AUTOSAR layered architecture ..................................................... 30
Figure 3.3: Interplay of safety and security in the automotive industry: now and future .......................................................... 33
Figure 4.1: Security Evaluation Framework ...................................................... 36
Figure 4.2: Flowchart of static analysis .............................................................. 37
Figure 4.3: Flowchart of code review ................................................................. 40
Figure 4.4: Fuzzing iteratively to ensure security testing ................................ 44
Figure 5.1: Software Under Test Architecture ............................................... 48
Figure 5.2: Scope of security testing ................................................................. 48
Figure 5.3: Adding a debugger to SUT and generating call graph .................... 50
Figure 5.4: Generate test cases and dispatch to SUT ....................................... 53
Figure 5.5: Results of fuzzing with valid-16 test cases before and after applying countermeasures ................................................... 57
Figure 5.6: Results of fuzzing with invalid test cases before and after applying countermeasures ................................................ 57
Figure 5.7: Data pipe presence in the software ................................................. 60
Figure 5.8: Error Propagation Scenario of segmentation fault ......................... 61
Figure 5.9: Countermeasure adoption and error value propagation .................. 62
List of Tables

Table 2.1: Comparison of the testing methods.................................................................13
Table 2.2: Banned family of unsafe functions and their replacements by SDL..................18
Table 2.3: Default pattern list..........................................................................................20
Table 2.4: A PLOVER example.......................................................................................21
Table 2.5: Seven Pernicious Kingdom.............................................................................21
Table 3.1: ISO 26262 Definitions of Automotive Systems Architecture..........................30
Table 3.2: Overview of AUTOSAR Layered Architecture..............................................31
Table 4.1: Overview of the attributes of Vulnerability & Threat Database....................38
Table 4.2: Obtaining attribute values in the Vulnerability & Threat Database................39
Table 4.3: Default pattern list with query “atoi” giving a pattern.....................................39
Table 4.4: Vulnerability and threat database after code review......................................41
Table 5.1: Comparison of combined methodologies in security testing........................64
Table 5.2: Feasibility study of security testing methods at different levels of abstraction....65
1 Introduction

“We are all now connected by the Internet, like neurons in a giant brain.”
- Stephen Hawking.

The world where we live today, is gradually associating us with the Internet and its related services. Internet is involving every single one of us with an urge to be connected together under one umbrella. As we are taking gradual leaps along with the technology, the increasing demand of seemingly more electronic and/or electrical (E/E) gadgets and instruments connected to the internet are practically becoming more obvious. One of such Internet-enabled services is a modern vehicle that is the mostly sought after technology, just after the evolution of internet-enabled smartphones. With the advent of Internet of Things [1, 2], it is being foreseen, that vehicles would be in a need to be connected together with smart devices like smart phones, smart cars, trucks, buses, Road Side Units (RSU) etc. in future.

![Figure 1.1: Needs for cyber vehicle security. [3]](image)

It is being predicted, to support innovative Internet related vehicle services such as adaptive cruise control, autonomous driving, crash avoidance system etc., there would be the necessity for Vehicle-to-X communications such as, Vehicle-to-Infrastructure, Vehicle-to-Vehicle, Vehicle-to-Mobile etc. (Figure 1.1) [3]. While Vehicle-to-X communication is a necessity, at the same time,
potential security issues associated with Vehicle-to-X communication are emerging rapidly. That is because Vehicle-to-X communication needs some constraints to be fulfilled, such as networking and software.

Networking and software based services (Figure 1.1) could achieve interaction between systems. The need of interaction between systems often leads to the requirement of additional software. Vehicle-to-X communication may bring in the scenario where multiple systems may be required to interact with each other. As a result, additional software would be needed to support such enormous interactions. Again, some software may be intertwined with other software that may need the other software to fulfill its operation. Eventually, the chain of interactions between systems and software would create a complexity explosion.

According to L. Boran, as the complexity of systems would increase, the associated risks would also increase. The associated risks would drive the necessity of more advanced security/data privacy solutions [4]. An example of an associated risk, as demonstrated by the researchers R. Verdult and F. D. Garcia, is car-theft. They demonstrated that by exploiting a security hole in the automotive software, a car-engine could be remotely started with the help of a software [5, 6]. If this happens, then it is assumed, unlocking of the doors remotely may also be possible. These are nevertheless worth-giving ideas but are some of the major risks and challenges that are threatening and must be addressed carefully.

On the other hand, the new risks can also give rise to new types of motivations that may drive an attacker to do malicious manipulations to the vehicles. These are financial gain, trading of privacy data, trading of vulnerability information of vehicle etc. [7, 8].

F.Y. Rashid points out, “Because many automotive systems are managed by tiny computers — from engines and brakes to navigation, air conditioning and windshield wipers — it may be only a matter of time before attackers exploit automotive software bugs to harm drivers” [9]. Software security bugs may be propagated into the in-vehicular network through untrusted interfaces and services such as unsecured wireless carriers, third party applications etc. Hence, the automotive platform software, which is the controller framework for the automotive systems, the hardware and the network components which are controlled by the platform software, would be at threat. Therefore, it is a dire necessity to discover the potential pathways of attack that attackers may adopt for exploiting the potential weaknesses in the automotive platform software and the associated
software components. Then appropriate countermeasures may be employed to prevent such attacks in the automotive software.

To delimit the exposure of potential weaknesses of software to the attackers, a systematic approach of security testing is required. The approach of finding and confirming those vulnerabilities that could have high chance of getting exploited, is the first requirement of the security testing. Then a systematic approach of adopting and applying appropriate countermeasures at critical software locations would be the most important goal of security testing.

1.1 Research Questions
The work addresses the following two research questions:

**Which methods and tools can be used for security testing and evaluation in the automotive software?**  
There are many methods and tools presently available for testing of software in the IT and automotive industry. Among them, some methods, for example, static analysis, fuzzing and code review are currently used for evaluation of security in software. These methods are not able to singly address security problems that might arise from the implementation phase. Therefore, an evaluation methodology is needed, to find which method/methods could be used for security testing and that would be able to address the security problems arising from the implementation phase in the automotive software.

In this thesis, a sample code that addressed a real world automotive application code, was used at the beginning to evaluate the identified methods for their feasibility analysis in the security testing of automotive software.

**How to combine various methods to systematically perform security testing?**  
In the automotive industry, the different methods required for security testing that had been used so far, followed ad-hoc approaches for security testing. Since the very beginning, there has been a need of a security evaluation framework that combines various methods systematically. A single method may always lack full proof strategy to eliminate all kinds of security problems. The solution is therefore, to utilize the advantages of other methods at the presence of one.

Therefore, a systematic approach to combine various methods should be beneficial. By systematic security testing, it may also be possible to prove that a potential vulnerability can be
exploited. To prove this, in this work we firstly identify the methods that can be applied for security testing. Then we propose a systematic approach for selecting the order in which the identified methods are used for security testing and try to find and confirm the vulnerabilities systematically by security testing.

1.2 Project Specific Goals

The major goal of this thesis is to improve the security strength in the automotive software. One way of achieving this goal is to perform systematic security testing of software. As a part of security testing, it is necessary to identify the potential vulnerabilities that may be exploited. Then, further investigation would be required to identify those vulnerabilities that have low impact and that may not affect the overall functionality of the program. By ignoring such low impact vulnerabilities, search space for finding high impact vulnerabilities would be effectively reduced. Then the identified high impact vulnerabilities should be further evaluated for security testing for confirmation of their impact. In the end, evaluation of results in security testing is required to identify the vulnerabilities that have been truly found exploitable in the experiment. Applying countermeasure should be the final goal to prevent those vulnerabilities from being exploited. In this way, vulnerabilities would be prevented with the help of security testing and this will improve security in software.

1.3 Research Methodology

The following steps (Figure 1.2) were used to achieve the security goals laid by the stakeholders:

Figure 1.2: Research methodology adapted from K. Peffers. [10]
According to K. Peffers [10]:

- The first phase is the Problem Identification & Motivation phase. In this work, it involves literature survey of the state-of-the-art research status as represented by the stakeholders in this project. Then a problem was defined, research questions were framed, and project specific goals were rationalized.

- The next phase is the Design and Development phase. In this work, this phase corresponds to proposing the design of a security evaluation framework for security testing of automotive software. It involves identification of methods for security testing and construction of security testing methodology.

- The next phase is Demonstration. In this work, this phase corresponds to the establishment of experimental set up and environment for security testing of the System Under Test (SUT). Then it involves carrying out security testing on SUT.

- The next phase is the Evaluation phase that comprise analysis and evaluation of experimental results. Finding appropriate countermeasure for the code through error propagation states and then applying the identified countermeasure on the SUT, were the next steps used in this phase. Then we change the design of the security evaluation framework and continue to perform security testing iteratively to confirm and eliminate security vulnerabilities.

- The next phase is the Communication phase that represents methods used to convey messages for lessons learned during this work. Presentation to a target audience and a report are the mediums used for communication.

### 1.4 Stakeholders

The project HEAVENS [11]: HEAling Vulnerabilities to ENhance Software Security and Safety, consists of the following partners: ARCCORE, Chalmers University of Technology, Combitech AB, Omegapoint AB, SP Technical Research Institute of Sweden, Volvo Cars and Volvo Technology Corporation (VTEC). The project HEAVENS is coordinated by VTEC, also known as Advanced Technology and Research within Volvo Group Trucks Technology, where this research work was mainly carried out. The goal of HEAVENS is to identify security vulnerabilities in software-intensive automotive systems and define methodologies along with tools for performing
software security testing. In this research work, the goal of HEAVENS has been adopted as the main goal.

1.5 Outline
The thesis is organized as follows. In Chapter 2, a background of the security concepts that are required for security testing has been introduced and explained in details their correlation with this work. At the end, the related works in security testing with the automotive domain as context, has been discussed. Chapter 3 presents the automotive background concepts used in HEAVENS. The concepts that are explained are automotive system architecture, automotive software, automotive functional safety standards, connection between safety and security, coding guideline standard, and key challenges of security in automotive software. In Chapter 4, the security evaluation framework for the security testing of software is presented. A discussion on static analysis, code review and fuzzing as security testing methods in the framework is presented. In this chapter, methods of finding vulnerabilities and threats in software, and then the methodology to confirm those vulnerabilities and threats using security testing, has been discussed. Chapter 5 presents the experimental set up and the experimental results of security testing. The results depict the changes observed in the vulnerability and threat database after every method is being used. Then the results are analyzed and countermeasures are applied. At the end we evaluate the results. Then Chapter 6 concludes our discussion with a summary and outlook of the evaluation and significant ideas for future improvements.
2 Security Concepts and Related Work

This chapter gives an overview of the concepts of security that we focus on this thesis. This includes techniques, concepts, methods and tools for testing of security in software. For definitions of terminologies that have been used in this thesis see Table A.1 [12] in Appendix A. In the first section, we define security testing and present the reasons for the need of security testing in automotive software. Here we introduce the automotive terms, Electronic Control Unit (ECU) and Controller Area Network (CAN). ECU is a computing unit that controls one or more of the electrical system or subsystems in a motor vehicle. CAN is a communication protocol for connecting electronic control modules in automotive applications.

Then we discuss the currently available security testing methods in the automotive industry. In the next section, in related work, we present recent examples of external and internal attacks in the automotive software that have been demonstrated. Thereafter, an example of a security framework that has been proposed by OEM’s and researchers of the related field, has been discussed. At the end, a brief summary of the related work and the proposed research topic, concludes this chapter.
2.1 Security Testing

According to NVD [13], CVE flaws generated from June 2005 – June 2015, as shown in the bar chart below,

![Total Matches By Year](chart.png)

**Figure 2.1: Total CVE flaws published on NVD from 2005 until 2015. [13]**

suggest that vulnerabilities are occurring in a relatively linear order. Since 2005, the number of vulnerabilities in software has increased. On average, between the years 2006 and 2013, the numbers are 5000 plus or minus 2000, every year. Most number of vulnerabilities, in the past 10 years came up in the year 2014. Through these years, software vendors like Microsoft, Google, Apple and Oracle have adopted security measures to restrict known and unknown vulnerabilities in software. However, it still remains a challenge to the IT industry as well as in other industries today how to find and eliminate vulnerabilities in software. Security testing has been commonly adopted as a countermeasure to cope with this challenge.

**What is security testing?**

Security testing is a process of engineering security in software that aims to find vulnerabilities and threats in software so that countermeasures could be correspondingly applied in order to prevent malicious attacks.

**Why is security testing needed at all?**

Security testing is done to assure that the system under test – SUT i.e., the software does not do what it is not supposed to do.
Security testing is also about making sure that countermeasures that are adopted to defend the vulnerabilities present in the software work correctly rather than that the intended functionality of the software works correctly. It has been found that most software is riddled by approximately 50-60% with flaws remaining unaddressed at the design stage and 40-50% vulnerabilities remain unaddressed at the code implementation stage [15].

Nowadays, security testing has become a necessary part in the software development life cycle (SDLC) as a countermeasure to check for faults and failure in the software. Software companies have started understanding the essence of security testing and they have started adopting measures such as threat modeling and risk analysis at the beginning stage of SDLC.

Comparatively, in the automotive industry, security testing is relatively unexplored. S. Bayer presents the methods of security testing that are available in the automotive domain, such as functional security testing, vulnerability scanning, fuzzing and penetration testing [16].

**Functional security testing** is used to investigate functional correctness and robustness testing of security functionalities. For example, an implementation of a cryptographic algorithm should be checked for its correctness. The implementations of cryptographic algorithms are often tested with official test vectors. As developers heavily rely on the specifications and official test vectors during code development, there remains some door of opportunity left for attackers to exploit the potential
vulnerabilities that may arise from other sorts of random test vectors. These kind of security vulnerabilities are missed out by functional security testing team. Adhering to MISRA C safety coding standards can reduce the number of such potential vulnerabilities in software.

**Vulnerability Scanning** is used to test the systems with known set of vulnerabilities that could be either unsafe functions or unsafe configurations. OpenVAS is a framework of vulnerability security scanning that particularly scans for open ports in the automotive software [17].

**Fuzzing** is used to test the systems with malformed inputs that might be able to uncover unsafe weaknesses and vulnerabilities. CAN messages are checked for integrity, availability, confidentiality, and freshness between inter ECU communication.

**Fuzzing techniques** as presented now, only investigate possible attacks through external ports and physical devices. The code for automotive software is not open source. Reverse engineering technique is presently used to retrieve the binary code and all types of security testing is performed with the help of third party debuggers, for example, *OllyDbg, IDA Pro* etc. Security testing, directly on the software source code has been out of scope for researchers. In this work, security testing is done on the source code of an automotive software.

**Penetration testing** is the sophisticated way of testing the whole system by a security tester with his knowledge of security testing. This might involve testing of software and hardware with single or combination of various security testing methods such as fuzzing, code review, manual inspection, static analysis etc.

2.2 Software Testing methods and Tools

This section describes the state-of-the-art software testing techniques, methods and tools that are currently in use with software related industries.

Here are three example techniques of software testing:

- **White box testing** - This technique is applicable when full knowledge of the target system architecture and the source-code is available.
- **Black box testing** - This technique is applicable when the source-code of the system under test is not available. The knowledge of internal architecture, configurations is absent in this technique.
- **Grey box testing** - This technique is applicable when the source-code is not available but some knowledge of the internal architecture is available.
This thesis uses white box techniques. Grey box or black box techniques are not used because the internal architecture and source code of the automotive software was available to us.

Static Analysis
Analysis that finds defects in the program without actually running the code is known as static analysis. Static analysis is a white box testing technique.

Intent
- Find defects and prevent run time errors
- Find security vulnerabilities

Tools
There are many tools available for static analysis. The tools that have been used in this thesis are rats, flawfinder, and Klocwork.

Code Review
Code review is the process of manually checking the source-code for security issues in a software. Code review is very efficient and accurate. Many serious security vulnerabilities that cannot be detected with any other form of analysis or testing, can be found by code review [18]. However, it requires a large amount of effort from human experts. Code review is a white box testing technique.

Intent
Issues that are particularly conducive to being found through source code reviews include concurrency problems, flawed business logic, access control problems, and cryptographic weaknesses as well as backdoors, Trojans, Easter eggs, time bombs, logic bombs etc. [18].

Fuzzing
Fuzzing is a negative software testing method that feeds a program, device or system with malformed and unexpected input data in order to find critical crash-level defects [19].
Fuzzing is a black box technique, but it may be used as white box testing, if needed (see chapter 5).

Intent
- Uncover common errors and potential vulnerabilities.
- Confirm security testing by exploiting identified issues.
**Tools**

There are many tools for fuzzing e.g., peach, sulley etc. These tools are very efficient, at the same time, very complex by nature. For simplicity, in this thesis work, an open source code [20] has been used that is serves our purpose of fuzzing. A debugger has been added to the system under test that helped to observe the results of fuzzing.

In the following table (see Table 2.1) we present the pros and cons of these software testing methods.
Table 2.1: Comparison of the testing methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| **Static Analysis** | • Does not run the code.  
• Identifies the exact line of occurrence of buffer overflow in the code.  
• Results can be used to find unknown vulnerabilities. | • Does not identify origin of the errors that may lead to an overflow vulnerability.  
• Unidentified source of errors remain dormant due to lack of all kinds of possible inputs on the code. |
| **Code Review** | • Gain relevance/insight of the probable exploitation of those vulnerabilities.  
• Helps to identify irrelevant vulnerabilities.  
• Completeness and effectiveness.  
• Accuracy can be achieved.  
• Fast for competent reviewers. | • Time consuming process.  
• Required manual debugging expertise.  
• Can miss issues in compiled libraries.  
• Cannot detect run-time errors easily. |
| **Fuzzing**   | • Uncovers unintentionally programmatical mistakes that could cause failure only at runtime.  
• Uncovers major sources of vulnerabilities as for example buffer overflows, value failures etc.  
• Uncovers unknown vulnerabilities | • Large number of test cases required to confirm a case of vulnerability.  
• Expertise required to generate intelligent test cases. |

In the above table (Table 2.1), the pros and cons of each method has been identified, and that could help to improve security in software (see marked rectangular boxes). Then those identified pros and cons has been further investigated to design the methodology of choosing the order of use of the testing methods in the security evaluation framework.
2.3 Secure Software Development Lifecycle

In the context of software security, it is been often said that “Software will always have security problems” [21] because security vulnerabilities are part by nature in the software. Security vulnerabilities in a software are produced from human errors that occur during the software development lifecycle (SDLC). It has been found that fixing these security vulnerabilities by adding security plugins or patches after the code has been deployed, would often require lots of rework. This rework would cost the company/individual extra man hours and extra time dedicated towards the client for overall maintenance of the software.

Therefore, a strategy for continuous improvement of security throughout the software development lifecycle, is required to ensure a secured product at the end that should be able to cope with challenges from attackers.

In fact, there are some strategies available that help to mitigate security vulnerabilities in the SDLC. Aberdeen group [22] presented three strategies that have been used by companies to combat the threats and vulnerabilities in software. These strategies are:

- **Find and fix**– This strategy uses application vulnerability scanning and penetration testing to identify the security vulnerabilities in the applications currently in production that needs to be addressed by the application developers.
- **Defend and defer**– This strategy uses web application firewalls or application-level proxies, to reduce or defer the need for security vulnerabilities to be addressed by the developers.

![Figure 2.3: Securing applications – Three high level strategies. [22]](image)
• **Secure at the source**– This strategy needs the integration of secure application development tools and practices into the software development lifecycle, to increase the elimination of security vulnerabilities before applications are deployed [22].

In this thesis, our goal is to secure software development and we use security testing to achieve this goal. Based on [23], we find two main approaches of security testing:

(a) Testing security mechanism to ensure that their functionality is properly implemented.

(b) Performing risk based security testing motivated by understanding and simulating the attacker’s approach.

We will focus on the second approach. We think that security testing should be included during SDLC to mitigate the security vulnerabilities. Since our work targets on existing code, therefore we choose to use “find and fix” strategy with special interest in ruthless checking of input invalidations. We started by performing security testing in the testing phase and ended up by applying the countermeasures in the implementation phase (see yellow outline in Figure 2.4). This implies that those specific countermeasures that we apply to the code using “find and fix” strategy, has to be applied exactly during the implementation phase in order to block that vulnerability permanently. This is why security testing is required during the implementation phase.

Practically when we perform security testing, we find the need to modify the code for applying countermeasure. To do so, we have to return to the implementation phase in order to apply the countermeasure. Then we switch back to the “find and fix” strategy to check if that countermeasure worked. In this way, by performing security testing iteratively, we are able to confirm a secure code that apparently prevents attacks from malformed inputs. We conclude that by adopting our approach of iterative security testing during implementation and testing phase, can eventually eliminate the security problems arising from malformed inputs.

Now let us compare our approach with the Microsoft’s Security Development Lifecycle (SDL) model [24] (see Figure 2.4). SDL is a widely used model by enterprises to identify threats and vulnerabilities in the SDLC. If we carefully look at this model by following the marked area with
blue outline, we can see that in the implementation phase, SDL uses mainly best practices of coding, tools that can analyze the written code and tools that can identify unsafe functions and instances. We also use these tools, but in addition we propose a new approach of analyzing the results from the static analysis tools. This new approach would help a security tester to identify the vulnerabilities and threats associated with the unsafe instances that result from the static analysis tools. Moreover, we also propose to use code review in this phase that would help the security tester to assign a relevance (HIGH/LOW) to the unsafe instances. We name it as the “exploitability” of the instance. That means, if in code review the exploitability is found to be HIGH, then that unsafe instance would have high impact in the running system, if exploited. By using static analysis and code review, we find that the search space of finding the vulnerabilities would be significantly reduced. The reason is because our approach is very systematic and accurate in finding the exploitability of the vulnerability, so the instances that this approach would find as “LOW” could be safely ignored, and thus theoretical search space is reduced. In the results, we will show that it is reduced.

Apart from static analysis and code review, we also propose to carry out fuzzing in the implementation phase. By fuzzing on the instances that have “HIGH” exploitability, which are found in code review, it is possible to confirm by security testing whether those instances are really attackable. If a coding team can find those instances beforehand, then an immediate countermeasure would save a lot of rework in this early testing phase. It can also avoid eventual patches or plugins when the code has been already delivered to the client. We also demonstrate that if by fuzzing on the known unsafe instance we are not able to identify the exact location of the source of vulnerability, then by identifying propagation of errors, with the help of repeated fuzzing, it is possible to find the exact location of the vulnerability and appropriate places for
countermeasures could be also identified. We demonstrate this with an example in chapter 5. Then in our framework, we also demonstrate how to find an appropriate countermeasure in the system under test.

The most interesting part with this proposal is that this entire security evaluation framework is systematic and flexible, such that, it can be used in the implementation phase and in the testing phase as well. Though we have not tested this principle by starting security testing during implementation phase, yet we predict that it is very much possible to perform security testing using the same methodology in this phase.

2.4 Guidelines to Safe Coding

In this section, we will discuss about best vulnerability avoidance schemes present in IT as well as other software related industries.

First we shall discuss briefly about **Common Weakness Enumeration** (CWE). Then we will describe the guidelines. CWE is a software community project that aims at creating a catalog of software weaknesses and vulnerabilities. The goal of the project is to understand flaws in software and create automated tools that can be used to identify, fix, and prevent those vulnerabilities [25].

![Figure 2.5: Types of vulnerabilities from 1999-2015.][26]

Now we will present the list of common vulnerabilities. The above excerpt of chart from CWE and NVD points out that the top three vulnerabilities of all time in software are arbitrary code execution, denial of service and buffer overflows. While arbitrary code execution is a special kind of code injection technique, it can originate from insufficient validation of input data. According to the National Vulnerability Database [27], the number of reported input validation vulnerabilities has increased from 25 (2%) in 2004 to 498 (10%) in 2013.
Denial of service could originate from the unhandled exceptions and bad practice of coding. Buffer overflow can also occur due to insufficient validation of input data. Buffer overflow is one of the most exploitable techniques that is often used by attackers to obtain elevation of privileges. Most vulnerabilities are commonly produced as a result of programming mistakes. Especially, particular mistakes in C language are for example using of unsafe functions, error prone pointer arithmetic, and lack of array index bounds of checking etc. As a result, programming errors such as NULL pointer dereferencing, buffer overflows, memory leaks etc. are thrown at run time during the execution of the software. Security mechanism in software is required to restrict run time failures. It can be done only with secure coding practices. The following sub-sections gives examples of safe coding practices in IT that should be practiced to address security problems for any kind of software.

2.4.1 Banned functions
SDL proposes a list of unsafe library functions in traditional code to be banned of use so that the threat of several kinds of overflow attacks could be reduced to some extent [28]. It also proposes us to use the safe library functions, developed by Microsoft, as replacement of those unsafe library functions. We mention this list in our work to adopt as an alternative countermeasure instead of range check, provided the unsafe instance fall under the function family shown in the table below (see Table 2.2):

<table>
<thead>
<tr>
<th>Function family</th>
<th>Replaced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy</td>
<td>strcpy_s</td>
</tr>
<tr>
<td>strncpy</td>
<td>strcpy_s</td>
</tr>
<tr>
<td>strcat</td>
<td>strcat_s</td>
</tr>
<tr>
<td>strncat</td>
<td>strcat_s</td>
</tr>
<tr>
<td>scanf</td>
<td>strncat_s</td>
</tr>
<tr>
<td>sprintf</td>
<td>sscanf_s</td>
</tr>
<tr>
<td>gets</td>
<td>gets_s</td>
</tr>
<tr>
<td>memcpy</td>
<td>memcpy_s</td>
</tr>
</tbody>
</table>
2.4.2 Validation of Input

“The first line of defense in a secure program is to check every untrusted input.” [29] We show in our work that lack of validation in input, introduces vulnerabilities in code that can be exploitable. For this reason, we need suitable measures during validation of the input. We discuss the common approaches of validation of input in the following sub sections:

Three approaches:

- Limit the attack surface in the program by decomposing it into pieces so that the attacker cannot communicate at all to most pieces including the communication path between pieces.
- Limit the types of inputs allowed to the exploitable areas.
- Ruthlessly check data on all input paths into the main program from all untrusted sources.

Example: If a number is expected in a program as input, there must be means to make sure that the acceptable data is in number format – typically one digit, at least. Regular expression might be used. E.g. ^[0-9]+$

In most situations the value to be accepted has a minimum or maximum. A check to ensure that a legal value is always accepted, should be made.

Using regular expressions to validate what to allow and what not as a string could be used to specify a legal string. E.g. ^[A-Za-z0-9]+$

A list of characters that can cause trouble [29]:

- 0 or NULL in C marks line termination character of a string.
- End line characters can be interpreted as command endings.
- Character with values higher than 127
- Meta-characters
- Characters that have special meaning in program
2.5 Vulnerability Analysis

In this section, we will introduce the concepts that are required for performing vulnerability analysis during the static analysis method.

2.5.1 Default Pattern List

First we use a default pattern list, which is a list of unsafe instances that can be either unsafe library functions, bad practices of coding etc. By using this list we investigate the security pitfalls in the code. We prepare a default pattern list, as shown in the table below (see Table 2.3), for which the first column represents the unsafe instance.

The second column assigns the unsafe instance a security attribute known as “pattern”, which is a type of a vulnerability in the code that could cause a threat to the vulnerability if exploited. For an example, if one of the unsafe instances “memcpy” or “strncpy” is used in the code, then there may be chance of occurrence of the “Buffer Overflow” vulnerability by some arbitrary random input. The result would be a segmentation fault that can threaten the system by putting it to a failure.

Table 2.3: Default pattern list.

<table>
<thead>
<tr>
<th>Unsafe Instance</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>• memcpy</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>• strncpy</td>
<td></td>
</tr>
<tr>
<td>• atoi</td>
<td>Integer Overflow</td>
</tr>
<tr>
<td>• Dereference</td>
<td>Pointer</td>
</tr>
<tr>
<td>• Use after free</td>
<td></td>
</tr>
<tr>
<td>• unused code</td>
<td>dead code</td>
</tr>
<tr>
<td>• commented code</td>
<td></td>
</tr>
</tbody>
</table>

2.5.2 Preliminary List of Vulnerability Examples for Researchers (PLOVER)

PLOVER is a conceptual framework that lists over 1400 real-world examples of vulnerabilities [30]. PLOVER defines a set of terms and concepts that could help in communicating about vulnerabilities at an abstract level. PLOVER is intended for use by parties who are interested in
vulnerability research and classification, including academic researchers, code auditing tool developers, secure programming researchers, and others (see Table 2.4).

Table 2.4: A PLOVER example.

<table>
<thead>
<tr>
<th>Sl. #</th>
<th>PLOVER Vulnerability Category</th>
<th>Kingdom</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[BUFF] Buffer overflows, format strings, etc.</td>
<td>1</td>
</tr>
</tbody>
</table>

In this thesis, we use the PLOVER list to identify the potential vulnerability on an unsafe instance. The potential vulnerability is identified with the help of an identified pattern, which is found in a previous step. This pattern acts as a query to the PLOVER list, which gives us a matching result from the PLOVER table.

2.5.3 Seven Pernicious Kingdom

K. Tsipenyuk, B. Chess and G. McGraw organized a taxonomy of common errors into a category of problems [31]. By using this taxonomy, developers would be able to understand the common security problems during code implementation that gives birth to vulnerabilities. They defined rules that helps to identify security vulnerabilities associated with different errors types along with the severity of consequences. The security issue falls into the Kingdom and the severity is termed as Relevance in the table, where 2 represents HIGH and 1 represents LOW. The taxonomy is adapted from 7 categories to 7 plus minus 2, and finally it has become stable at 7 plus 1 as shown in the table below (Table 2.5):

Table 2.5: Seven Pernicious Kingdom. [31]

<table>
<thead>
<tr>
<th>Sl. #</th>
<th>Kingdom</th>
<th>Relevance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input validation and representation</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>API abuse</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Security features</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Time and state</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Errors</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Code quality</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Encapsulation</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Environment</td>
<td>1</td>
</tr>
</tbody>
</table>
In this thesis, we have used the above mentioned taxonomy (Table 2.5). The first row, representing “Input Validation and representation” as Kingdom number 1, has been used by us as an input to a result in the code review method of the security evaluation framework. This helped us to decide on the exploitability of a previously identified potential vulnerability.

2.6 Threat Analysis

Microsoft proposed a threat analysis model STRIDE [32] to identify the threats based on the goal and purpose of the attacker. We use this model in our framework to identify the potential threats in software.

Definition of STRIDE [32]

- **Spoofing** attackers pretend to be someone or something else
- **Tampering** attackers change data in transit or in a data store
- **Repudiation** attackers perform actions that cannot be traced back to them
- **Information Disclosure** attackers get access to data in transit or in a data store
- **Denial of Service** attackers interrupt a system’s legitimate operation
- **Elevation of Privilege** attackers perform actions they are not authorized to perform

We have used STRIDE to identify potential threats in the static analysis method and in the analysis of results.

2.7 Related Work

Security testing can be performed in two ways. One way is through external attacks that involves physical interface with a device connected to the vehicle. The other way is through internal attacks that uses software based interfaces, such as another software or network protocol. External attacks involve finding backdoors to the internal network through open ports for example, On Board Diagnostic 2 port. In computer security, these ports are traceable through a process known as Vulnerability Scanning. Attackers tend to reap attack vectors by gathering weaknesses of the targeted systems through information gathering over the internet. Information gathered through this process could be host network information, IP addresses of the targeted systems, open ports, firewalls or even network gateways or router information. Whereas, internal security testing may be executed by gaining authorization of certain services through exploitation of software security
vulnerabilities or by elevation of access privileges. Elevation of access privilege could also be obtained through software exploits, social engineering or by getting confidential access data from the insiders.

2.7.1 External Attacks on automotive software
This section presents some of the external attacks to the in-vehicular networks and software that researchers have conducted and verified.

Compromised ECU
In the automotive domain, external attacks could happen by sending arbitrary messages to the CAN bus through the On-Board Diagnostics (OBD) 2 port. A team of researchers, has demonstrated that infiltrated ECU’s could be used to enhance a specific functionality of a vehicle [33]. They sent arbitrary messages to the CAN bus through the OBD 2 port, sniffed the content of those messages and then by reverse engineering dumped the modified code of the ECU through a third party debugger. They showed, a wide array of attacks could be also performed, such as open and close doors, engage or disengage breaks, disturb engine functions, taking control of the radio, etc. These attacks are apparently due to lack of internal security in the software, networks and protocols.

Attacks on tire pressure monitoring system
Remote wireless attacks could be possible through radio communication. The idea was proved by Rouf et al. [34] who exploited the tire pressure monitoring system that acts between the vehicle and the sensors present in the tires of a vehicle. The tire pressure monitoring system equips each tire with a sensor that has a unique 32 bit identifier. Every vehicle recognizes its own wireless sensors from this 32 bit identifier whereas it rejects all messages from other vehicles.

The radio communication between the tires and the vehicle was insecure. Any low cost antenna or low power amplifier around 40 meters nearby the vehicle, could also catch the radio signals. By taking this advantage of the insecure channel, the researchers exploited the tire pressure monitoring system with spoofed messages. They showed that it was possible to accept all messages with correct ID’s that triggered many warning messages at the same time to the driver. The researchers also managed to crash the ECU that received the messages. To fix the problem the vehicle had to be taken to the repair shop.
2.7.2 Internal Attacks on automotive software

In this section we will explain an example of internal attack in automotive ECU.

Attacks on media player

It is an example of an internal attack that got access of the CAN bus through a vulnerability that was found in the media player software by the same group of researchers [33]. The media player software was reportedly from a third party software provider. The media player software was able to get access to the CAN bus when a file of type CD, Mp3 or WMA was played.

The researchers found that, a buffer overflow exploit in the software was possible to be launched when they crafted a CD with a malicious attack vector. As soon as the CD was played the buffer overflow exploit occurred on the media player software which then sent arbitrary messages to the CAN bus. In future, it would not be difficult for attackers to launch an attack of this kind, if this music player is connected to the internet through untrusted wireless networks. Either the wireless network could be misused or the software that plays the music player could be exploited. In fact someone in the vehicle downloading a carefully crafted malicious music while driving, and playing that music could immediately put the vehicle at threat. If the intention of the attacker is as harmful as damaging any crucial engine functionality, then it might not be difficult for an attacker to find a skillful way to do it. Then the consequences could be road accidents, which is also a threat to physical safety.

2.7.3 Other External and Internal Attacks

A 24-year old researcher, Eric Evenchick claimed that he had prepared a device known as CANtact that could be connected on one end to a computer’s USB port, and on the other to a car or truck’s OBD 2 port to make car hacking easier and cheaper [35].

“The average coder isn’t familiar with the protocol most cars’ computers rely on to communicate. But Evenchick has written open source software for CANtact that automates much of the manual work of CAN bus hacking. Like the earlier work by Valasek and Miller [35], the CANtact is designed to send commands in Unified Diagnostics Services, the CAN protocol that auto mechanics use to communicate with electronic control units (or ECUs) throughout a vehicle. That allows anyone to write python scripts that can automatically trigger commands in a car’s
digital network that range from turning off its “check engine” light to automatically pumping its brakes” [35].

The same group of security experts, Charlie Miller and Chris Valasek, as reported by Wired [36], took control of a Jeep Cherokee running at 70 mph alongside its driver, Greenberg, from Miller’s living room. “They accessed the Jeep's computer brain through its “Uconnect” infotainment system and rewrote the firmware to plant their malicious code. Once in, the duo began blasting hip-hop through the stereo system, turned the AC to maximum and, ultimately, killed the transmission and brakes” [37]. This is an example of internal attack because the researchers exploited the wireless network and the software interface of the Infotainment system to attack the vehicle.

Other types of attacks that attackers are targeting are for example authoritative functionalities in the vehicle. For instance, it is profitable for an attacker to reduce the value of the traveled distance from odometer logs while selling a car. In Germany, according to police investigations, around 2 million cars are subject to odometer manipulation per year with an average loss per vehicle of around 3000 € resulting in total losses of around 6 billion € per year [38]. Furthermore, critical data that are more attractive to the attackers are the data stored in the ECUs such as crash data, data for insurances, or warranty indicators. Malicious code crafted with negative or extremely large inputs can be used to systematically launch an attack on these data values. For example, data such as vehicle speed, seat belt status, brake pedal position etc. are typically recorded in the seconds before a crash. A driver who has been involved in an accident could be motivated to change the recorded data to indicate that the brakes were applied when they really were not [38].

All the above proven attack methods led by the researchers suggest that there are definitive weaknesses either in the protocols, software from the legal third parties, software from the vendors and the other in-vehicle network weaknesses could be leveraged by the attacker by launching attacks both physically through OBD hardware tools and even remotely through wireless networks and at the presence of the internet.

2.7.4 Security Framework
Here are some of the ongoing research of countermeasures at the level of software to enhance security features that are required for connected vehicles.
Secure Onboard Communication

According to the recent specification [39] of Secure Onboard Communication (SecOC) from AUTOSAR, the fully implemented SecOC module should provide necessary authentication mechanisms to secure end to end communication between ECU’s. This mechanism should check data authenticity, data integrity and data freshness in inter ECU communication which should improve inter ECU communication in a safe and secure way. The protection of authenticity, integrity and freshness of data shall be applicable for several different types of data, messages. The use cases that define the objectives of such protection mechanism are summarised below:

- Messages directly used for authentication – e.g., message carrying VIN – Vehicle Identification Number should be protected from attacks.
- Safety relevant messages – e.g., Brake, Speed, and Torque request should be protected from attacks.
- Real-time critical messages – e.g., the big majority of the sensors communication in powertrain, messages related to vehicle dynamics should be protected from attacks.
- Different security levels / requirements
- End-to-end and Point-to-point protection
- Highly frequent data, transmitted on a time or event triggered schedule need protection.
- Adding authentication information to an existing message
- Entity authentication
- Intrusion detection
- Backwards compatibility with existing AUTOSAR solutions and tools

The proposed solution will fit in existing AUTOSAR solutions and tool chains on hand.

Summary

From the literature survey, it is seen that there is a momentum shift in vehicle technology going on that would be enable vehicles to communicate to a wide spectrum of connected vehicles and devices in an infrastructure. Such an infrastructure could be connected through trusted and untrusted carriers and networks that might leave potential weakness to attackers. As such, software would be downloaded, updated and executed on these vehicles. Messages would be exchanged between vehicles with the help of software through unsecured networks, leaving attackers a lot of
opportunity to exploit their weaknesses. Attackers could try to find tricks to read a software from ECU. One way of doing it is dumping the ECU software through reverse engineering. After reading the code one can make malicious fabrications in the code that may trigger attacker’s running scripts. Then the attackers may choose to threaten the vulnerable system by denial of service attacks. Or they might be able to perform fuzzing on the systems, creating the attack vector like buffer overflow. Using buffer overflow the attacker might be able to write anything in the ECU memory. The attacker can also exploit the buffer overflow to alter the instruction sequence execution.

Current research, as discussed earlier (see section 2.7.1), is going on how attacks can be carried out on these systems externally, which leads researchers to think like an attacker as if he has no knowledge inside the box. Then the attacker finds some way to read and change the code and reflash the software into the ECU again. Then the attacker finds what kinds of new attacks are possible after the modified software keeps running inside the ECU.

In another easier way of attack that a smart attacker could launch, is to flood these E/E systems with smart or random inputs that could automatically expose the vulnerabilities in the system. Then the system under test may behave abnormally after the exploit. Using this knowledge the attacker might be further motivated to spawn a new way of attack. This method has been found implemented with the help of a fuzzing tool namely beSTORM connected with the physical CAN bus, simulated by an ECU simulator namely, CANbuster [40]. However, this is a closed tool and out of reach to the research community. The tools and devices that are required for this approach are only available to manufacturers, enterprises and to the governments.

So far, the investigation with fuzzing to the automotive E/E systems involved the CAN protocol and physical CAN bus. Fuzzing attempts have been made inside the ECU software from the outside. This thesis work assumes that if the attacks happen from outside, then what kind of security principles should be adopted by Original Equipment Manufacturers (OEM) in the code development stage to prevent such attacks.

There are presently methods and frameworks available that adopts the strategies of fault injection for robustness testing on AUTOSAR based systems [41]. These methods evaluate robustness on software components to find possible weaknesses in automotive software [42]. But, they never addressed propagation of errors, which we do.
This thesis work follows the strategy of fuzzing at the software component level, without involvement of physical devices, which has not been investigated so far in the automotive software. This work also attempts to propose and implement a flexible framework that could be used in a portable way to different abstraction levels of the E/E systems. The framework could be repetitively used to uncover known and unknown vulnerabilities during the implementation and testing phase.

3 Automotive Concepts

This chapter provides a background to the technologies and concepts used in the automotive industry. It presents an overview of the key automotive concepts that are used in the HEAVENS project. The first two sections discuss Automotive Electronic/Electrical (E/E) system [43] architecture and the AUTOSAR platform respectively. Next, we discuss the need of functional safety practices standard, ISO26262, for road safety. Then the connection between safety and security is explained. Thereafter, the clear distinction of the present state of safety, functional safety and security in the automotive industry is depicted through a diagram. In that diagram, the fusion of safety, functional safety and security as a need of future, for overall automotive security, is also shown. Next, we introduce the coding standard, MISRA C, for safe practices of coding. We conclude this chapter with an introduction to some of the common challenges of the software security perspective in the automotive industry.
3.1 Automotive Electrical and Electronic (E/E) System Architecture

In this section, we present the E/E architecture that can be described as a set of multiple E/E systems, where every E/E is composed of a set of Electronic Control Unit (ECU) software, known as an item. A single ECU software is represented as a System. A system is made of one or more software components where a software component is made of a set of files. The atomic level of a software component is the software unit, which can be made of a single file with a single function its lowest atomic level. The following figure depicts an E/E system (Figure 3.1) with an example at every level:

![Diagram of E/E system architecture]

**Examples**
- (Cruise control)
- (Set of ECU Software)
- (Single ECU Software)
- (Set of C Files)
- (A file of single C function)

**Figure 3.1:** E/E automotive systems architecture [43].

According to ISO 26262 [43]:

- An element has been defined as a system or part of a system including components, hardware, hardware parts, and software units.
- A component has been defined as a non-system-level element that is logically and technically separable and is comprised of more than one hardware part or of one or more software units.
- A failure is the termination of the ability of an element, to perform a function as required.
- A fault is an abnormal condition that can cause an element or an item to fail.
Table 3.1: ISO 26262 Definitions of Automotive Systems Architecture. [43]

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Unit</td>
<td>Atomic level Software Component of the Software Architecture that can be subjected to standalone testing</td>
</tr>
<tr>
<td>Software Component</td>
<td>one or more Software Units</td>
</tr>
<tr>
<td>System</td>
<td>Set of elements that relates at least a sensor, a controller and an actuator with one another.</td>
</tr>
<tr>
<td>Item</td>
<td>System or array of systems to implement a function at the vehicle level, to which ISO 26262 is applied.</td>
</tr>
<tr>
<td>E/E System</td>
<td>An E/E System is defined as a system that consists of electrical and/or electronic elements, including programmable electronic element.</td>
</tr>
</tbody>
</table>

3.2 Automotive Software

In this section, we discuss about the software architecture that are currently used in automotive software. The HEAVENS project uses AUTOSAR as the platform architecture (see Figure 3.2) for the automotive software, which we also use. The description of different layers of this architecture is elaborated in the table below (see Table 3.2).

Figure 3.2: AUTOSAR layered architecture. [44]
Table 3.2: Overview of AUTOSAR Layered Architecture.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Overview</th>
</tr>
</thead>
</table>
| Application Layer    | **Description:** All applications residing at the top layer is known as the application layer. A software component (SWC) is the smallest unit of the application layer.  
**Purpose:** Every SWC are intended to perform a specific task. AUTOSAR contains specific interfaces to combine multiple SWC into a single application. |
| RTE                  | **Description:** Middleware layer providing communication services for AUTOSAR software components (SWC) and applications containing AUTOSAR sensor/actuator parts [45].  
**Purpose:** Make AUTOSAR software components (SWC) independent of mapping to specific ECU [45]. |
| Basic Software       | **Description:** Basic software layer is a standardized software layer without functionality of its own. It is divided into three sub-layers as described below.  
**Purpose:** This layer offers both hardware dependent and hardware independent services to higher layers using application programming interfaces. |
| Microcontroller      | **Abstraction Layer (MCAL)** is the lowest sub layer that makes higher software layers independent of the microcontroller.  
**The ECU Abstraction Layer** is the middle sub layer that interfaces the drivers of the Microcontroller Abstraction Layer and makes higher software layers independent of ECU hardware layout.  
**The Services Layer** is the highest layer of the Basic Software that provides basic services for application and basic software modules [46]. |
| ECU Resources        | **Description:** The lower most layer is the Microcontroller layer also known as MCU, which contains the ECU hardware [47].  
**Purpose:** It provides services for basic microcontroller initialization, power down functionality, reset and microcontroller specific functions [48]. |

3.3 Functional Safety Standard

The automotive industry is familiar with the functional safety standard ISO 26262 [43]. OEM and automotive suppliers have designed system processes adhering to the functional safety standard ISO 26262. By following this standard, will help automotive suppliers to understand and implement cybersecurity processes. This will also help to maintain a consistent interplay between safety and cybersecurity processes. It will allow to implement safety mechanism and countermeasures for security vulnerabilities.

While most standards are devised for the IT industry, ISO 26262 is the only standard that has been specifically devised for the Electrical/Electronic Control Systems in the automotive industry.
Elements from other security process were identified to be compatible with the design for cybersecurity processes in automotive systems e.g., Software Development Lifecycle (SDL).

3.4 Automotive Safety and Security

“Safety features are intended to protect against technical failures whereas security features are expected to protect against malicious manipulations” [49]. In this section we discuss the relation and reasons of interplay between safety and security in the automotive industry. We also illustrate the present state of safety and security in the automotive industry through a diagram. In this diagram we also illustrate the need of an integrated approach for safety, functional safety and security in future. In the next two sections we discuss safety coding guideline standard and future challenges in the automotive industry,

3.4.1 Interplay between Safety and Security

Since software plays a major role today in controlling electrical and electronic machines and devices, it is also a major contribution towards the technology advancements. Automotive vehicle innovations of today is one such example that are mostly driven by software. However, safety is one of the key aspects that is thoroughly looked before an automotive vehicle is launched and sold in the consumer market. On the contrary, software security is another aspect that had been given less attentions so far in the automotive industry.

Software security in vehicles can protect human being from physical hazards. Attackers could try to manipulate the software to make the vehicle fall into failure. Researchers have demonstrated that an attacker could launch an attack by controlling the vehicle functions like brakes, lights, engine etc. [33]. An attacker can also put the headlights or horns on remotely or disengage brakes so that the vehicle could not be stopped. In 2013, for instance, Darpa-funded security researchers, Chris Valasek and Charlie Miller showed that it was possible to send digital commands from a laptop connected to a car’s CAN bus that affected steering, slammed on brakes, or even disabled brakes at some speeds [35]. Security in software is required to prevent these malicious manipulations so that known attacks could be relatively avoided.
The above figure describes that safety, functional safety and security are needed in an integrated approach in future. In this work, the focus was to look for failures specifically caused due to security failures.

3.4.2 Coding Guidelines

MISRA C is a standard [51] that was developed to provide safety coding guidelines for embedded systems as well as for all kinds of E/E systems. As a natural choice, the automotive industry adopts this standard for their E/E vehicle systems that describes safe and secure practices of coding for road safety. However, MISRA C guidelines have not been properly followed in the automotive ECU software previously, at least found true in the Toyota investigation case [52].

An instance of exception is encountered in this work when the system under test crashed due to unhandled divide-by-zero exception, which violated one of the MISRA C guidelines.

3.4.3 Challenges of Automotive Software Security

This section presents an outline of the security challenges [53] that would be obvious as upcoming issues with automotive software.

**Figure 3.3: Interplay of safety and security in the automotive industry: now and future. [50]**

The above figure describes that safety, functional safety and security are needed in an integrated approach in future. In this work, the focus was to look for failures specifically caused due to security failures.
ECU Software

Vehicles of today consist of more than 100 ECU’s [54] and the software that runs all the internal communications consists of hundreds of millions of lines [55] of code. As of today, there is lack of security in the in-vehicle network of automotive domain. After 10 to 15 years the same vehicles would run and it is expected that security would be improved and there would be Vehicle-to-Infrastructure communication, Vehicle-to-Vehicle communication and Vehicle-to-Mobile communication. It leads us to a question, comparing the present computation power of the ECU’s would it be good enough to deal with the security problems with same computation power and same number of ECU’s of today?

In the computer domain, we have seen the software solution has always proved to be a cheaper solution in comparison with increasing silicon area in terms of increasing the computation power. More computation power may be required to perform certain cryptographic hash functions, encryption and decryption, secure authentication protocols etc. If more number of ECU’s are used to increase the computation power, then it is surely expected to increase the expenses to the companies which could affect them on a large scale. They would naturally look for software solutions that can purposely fit the security requirements in a cheaper way. In a survey by Aberdeen Group, it was found companies are indeed getting 4.0 times return of their annual investments by using secure by design principle in software development [22]. It is not far that the same principle would be used in the automotive industry to bring down the cost of security in vehicle at an affordable price.

Operating System (OS) level security

Features pertaining to OS level security has hardly been addressed in the automotive domain. ASLR is a technique that is widely used these days in the computer security area to secure the OS. According to S. Checkoway, Address Space Layout Randomization (ASLR) should be encouraged to be used as it should be possible to be implemented easily on simple processors and it also increases the attack burden for potential attackers [56]. But ASLR also needs enough memory space on ECU which is arguably not feasible at the moment.

N^X or randomized base address is another method that is used to prevent execution of base address of the memory regions under buffer overflow attacks in the operating systems of computers. It means telling the CPU via page protection flags that it's not allowed to execute
instructions from the stack. It has not been tested in the automotive domain if the same mechanism of protection could work at the OS level in the ECU’s.

In computer security, “return-to-libc” is another way of attacking the OS that beats the N^X protection. It has not been tested in the automotive OS, the only reason of it being more advanced and sophisticated technique and infeasible at the moment in automotive domain.

Role based access control, authentication, authorization to services such as firmware authentication for firmware version identification, firmware updates over the air (FOTA), are, missing in the current context of the automotive OS [57].

Third party software

In the near future, it is not hard to imagine that there would be an online store available to the vehicle owners, similar to the Google Play or the iOS App store that would contain third party applications and services ready to be installed for vehicles. In a venture from Google, Android Auto apps [58] has been launched and apps are being developed in the Android car segment. Tools required for deploying those apps has been made publicly available to the android developers worldwide. It is natural to think that there would be many developers worldwide who would be developing the automotive software have zero or less knowledge on software security and therefore the source code that the developer might write would lack security principles like secure by design. Someone downloading and using that software in his vehicle may make his internal vehicle network vulnerable thereby allowing interested attackers to sniff on their vehicular networks and perform malicious attacks such as arbitrary code execution, denial of service, buffer overflow etc. Therefore to verify those application services from the third party vendors in a secured way, a standardized secure architecture is required that could run those unsafe services in a safe and secure environment and yet not disturb the basic functionality of the vehicle. There are many proposed solutions as of date today with the desktop computers, such as virtualization, sandboxing, trusted computing platform, issue of certificates from a certifying authority for validity of software and principles of least privilege. None of these have been implemented so far in the ECU software and it remains a tough challenge to the researchers to implement these security mechanisms in the automotive domain as the same challenge that is being faced today with the other handheld smart devices.
In this chapter, we shall discuss the security framework that we used in this thesis work. The framework consists of three methods that are used, one after another, in order to evaluate security in software (see Figure 4.1). The first method is Static Analysis. This method takes source code as the input and produces a list of unsafe instances as output. By using the results given by the static analysis tool, we take it further to identify the vulnerabilities and threats associated with the unsafe instances. The second method is Code Review. This method takes the result of static analysis as the input and gives the relevance, termed as “exploitability” in this thesis, as the output. The third method is Fuzzing. This method takes the results of code review as input and finally gives the result of exploitation confirmation as output.
As discussed in Chapter 2, in existing models (SDL), fuzzing or any other dynamic analysis techniques are applied directly after static analysis. As seen above, however, we inject code review in between static analysis and fuzzing. While static analysis gives us result of unsafe instances in our framework, code review acts as scrutinizer on those results. Code review will help us to find the root cause of the vulnerability in the code and would enable us to decide on the instances to investigate. As a result, we think that it will reduce the search space of the vulnerabilities in the code.

Fuzzing and repeated fuzzing would help us to check if those vulnerabilities can be exploited in real, and if yes, then we need to apply countermeasures for them. To check if our countermeasure works, we need repeated fuzzing to run our exploitation experiment.

4.1 Static Analysis

Static analysis is a method of analyzing the code without running it, as clearly discussed in chapter 2. In static analysis, first the static analysis tools rats, flawfinder, and klocwork generate their respective reports of the occurrences of the unsafe instances in the code with a line number and potential impact or a suggestion of the mitigation that could possibly help to avoid the major threat.

4.1.1 Vulnerability Analysis

As mentioned in Chapter 2 (section 2.5), we start our method of vulnerability analysis on the results of static analysis.
Vulnerability & Threat Database

As the first step, we construct a vulnerability and threat database. We use this database in order to make it easier for us to analyze the vulnerability and threats and record the results of security testing offline. We also store the status messages to the unsafe instances in the database while performing security testing. A security tester can identify the potential threat using the knowledge gained from the different attribute values associated with the unsafe instance. In the vulnerability and threat database, we insert the identified unsafe instances as one key attribute of a record, and identify the other attributes like pattern from a default pattern list, potential vulnerability and kingdom number from the PLOVER list [30], and potential threats associated to the instance. All of these attribute values could be saved as a record to the unsafe instance in the database. The table below (Table 4.1) depicts the attributes of a record in the vulnerability and threat database.

Table 4.1: Overview of the attributes of Vulnerability & Threat Database.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Instance</th>
<th>Potential Impact</th>
<th>Pattern</th>
<th>Potential Vulnerability</th>
<th>Kingdom</th>
<th>Potential Threat</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To start vulnerability analysis, first we prepare a default pattern list (see section 2.5.1 & Table 2.3). Then using the pattern we identify the other associated security attributes (see Table 4.2) including potential vulnerability, kingdom and potential threat.
Table 4.2: Obtaining attribute values in the Vulnerability & Threat Database.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Which</td>
<td>What</td>
<td>How to spot a pattern?</td>
<td>What could be the potential vulnerability?</td>
<td>1</td>
<td>How could it affect?</td>
</tr>
</tbody>
</table>

atoi, main.c, 30

Unless checked large numbers can roll over to negative numbers

Integer Overflow

[BUFF] Buffer Overflow

Tampering of Data / Denial Of Service

As illustrated in the above table (see Table 4.2), a query to the Vulnerability & Threat Database with the unsafe instance “atoi” gives the resulting pattern “Integer Overflow”. The pattern is obtained by querying to the Default Pattern List with “atoi” as shown below (see Table 4.3).

Table 4.3: Default pattern list with query “atoi” giving a pattern.

<table>
<thead>
<tr>
<th>Unsafe Instance</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>• memcpyn</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>• strcpyn</td>
<td></td>
</tr>
<tr>
<td>• atoi</td>
<td>Integer Overflow</td>
</tr>
<tr>
<td>• Dereference</td>
<td>Pointer</td>
</tr>
<tr>
<td>• Use after free</td>
<td></td>
</tr>
<tr>
<td>• unused code</td>
<td>dead code</td>
</tr>
<tr>
<td>• commented code</td>
<td></td>
</tr>
</tbody>
</table>
PLOVER gives us a list of the vulnerability types that are very common in the security testing world, as discussed in chapter 2 (see section 2.5.2). After we have obtained the pattern, which is “Integer Overflow” in this case, we then map it to PLOVER [30]. This mapping gives us the potential vulnerability and two more important results which we get by combining PLOVER and “Seven Pernicious Kingdoms” [31] together. From the obtained result we can make a brief idea of the type of the threat that can be imposed by this potential vulnerability.

The first result is the potential vulnerability “[BUFF] Buffer Overflow”, second is the Kingdom “1” to which the pattern belongs and third, the relevance “HIGH”. We further looked into Kingdom 1 in “Seven Pernicious Kingdoms” list [31], and it revealed us that the major cause of errors and corresponding failures for this type of pattern is the input validation (see Table 2.5), the relevance of which is 2, meaning HIGH.

Research reveals that using invalidated input as a part of directive or command to a subsystem can introduce vulnerability [59]. In our case, this hypothesis turns out to be true.

4.1.2 Threat Analysis

Next step is threat analysis. We identify the potential threat by using the knowledge of all the attributes received so far from the results of vulnerability analysis. To find the potential threat, we followed a generalized approach from the Microsoft SDL model that we use in our framework. As mentioned in section 2.5.3, the approach is about mapping our identified threats with the help of STRIDE [32] model.

We map the potential threat to the STRIDE [32] model (see Figure 4.1) in order to predict the extent of the attack that can be made.

4.2 Code review

![Flow chart of code review](image-url)
As discussed in Chapter 2 (section 2.2), Code review is a method of manually reviewing the code. It is very accurate based on the human knowledge, but also require a significant amount of human hours. After we have identified the potential vulnerabilities and threats, our next job is to identify whether the attack is even possible. To gain some security insight of those instances, we carried out a deep manual review in the code.

In this method, we analyze the exploitability of the unsafe instances, which needs the results of static analysis as input. Exploitability can be measured as either HIGH or LOW. We trace the instance in the code and look for the dependent variable or functions that receive and transfer any kind of input to this instance. By tracing such variables or functions backwards with the help of the call graph, we may be able to find where the input for this unsafe instance originate. If the input is passed from the source through functions and variables as constants variables, then it may be considered as less vulnerable and therefore may be ignored and marked as LOW.

If there is no protection since the beginning from where the input flows into the code until it reaches the unsafe instance, then that input can become a cause for a potential vulnerability. It may pose different kinds of threats to the intermediate variables and functions that it would pass through during its flow until it reaches the identified instance. This kind of threat mainly exploits buffer overflows, integer overflows, as for instance. So, we mark this unsafe instance as “HIGH”.

Table 4.4: Vulnerability & Threat Database after code review.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Instance</th>
<th>Potential Impact</th>
<th>Pattern</th>
<th>Potential Vulnerability</th>
<th>Kingdom</th>
<th>Potential Threat</th>
<th>Exploitability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flawfinder</td>
<td>atoi, main.c, 30</td>
<td>Unless checked large numbers can roll over to negative numbers</td>
<td>Integer Overflow</td>
<td>[BUFF] Buffer Overflow</td>
<td>Tampering of Data/ Denial Of Service</td>
<td>HIGH/LOW</td>
<td>How to assess?</td>
</tr>
</tbody>
</table>

In general, code review is a rigorous technique that helps a security tester to test his knowledge and expertise to identify the sources and points of attack in the code.

If the software is made of millions of lines of code, then obviously it is not infeasible as a business goal to use code review as a method to manually identify every unsafe instance and measure its exploitability. Again, it would not be feasible to trace an instance for the original source of its inputs by traversing it backwards, for example 1000 files and 5000 functions, which is
practically impossible at the moment by performing manual code review. Normally, all such propagation of the vulnerable inputs would never cause all functions or variables to fail at the same time. Therefore, sometimes, some inputs would be of less concern if it did generate failure value but there was a very minor impact that may be tolerated.

Therefore, some vulnerabilities that are of immediate concern and that can be easily investigated for input verification, could be used as a good starting point for further investigation. Hence, investigation could be initiated by using the instances from the results of static analysis that can be traced back using caller and called function relation.

Notably, code review on the results of static analysis would be benefitted according to this approach, because static analysis may give some results of the vulnerable unsafe instances in the code, some of which may appear as a stage in the data pipe (see 5.3 Observation II). By using code review according to this approach, we found that search space for finding vulnerabilities is reduced drastically. We also found that using this method would also help to find unknown vulnerabilities that would cause different kinds of attacks.

So code review on the results of the static analysis first finds the root cause of the vulnerability that originate from input invalidation. Then it helps to reduce the number of unsafe instances by rating their exploitability. Then using those few instances for deep investigation would also result in finding software bugs and unknown vulnerabilities.

4.3 Fuzzing
Fuzzing is another method of software security testing that helps to uncover vulnerabilities in the software, which is exploited by running with valid or invalid random inputs.

4.3.1 White Box Fuzzing
If white box fuzzing is the only method that is used to identify vulnerabilities in software, then it might be possible to identify some vulnerabilities by employing rigorous debugging techniques. Then that would become very much costly because a lot of effort would be required to identify those vulnerabilities even though the software may contain very few vulnerabilities. Therefore, employing such a technique may not be cost-effective for a very small software. Again, in white box fuzzing as a sole method, if debugging measure is not accurate then it might happen that it should cost only a single iteration of fuzzing to find out a single cause of failure. Debugging is an art and it needs a lot of understanding of the code and expertise to experiment the code. Sometimes
testers unintentionally miss out important debug statements that would be very important in the context of finding the software bug.

For a large software, fuzzing as a single method in security testing can turn out to be costly. Because, in this case, the software will have to include debugging measures that had been missed out in the previous fuzzing iteration by the security tester. This should result into multiple white box fuzzing iterations.

4.3.2 Black Box Fuzzing
If the code is not open source i.e. only black box fuzzing method is used, then according to us, it would become relatively difficult where to find in the code those vulnerabilities that were actually exploited. But it may be possible to the other researchers of this field at the same time.

4.3.3 White Box Fuzzing and Security Testing Methods – A Combined Approach
If the code is open source, which is, in our case, it may be possible to find vulnerabilities in the code in an efficient way by combining security testing methods and tools with fuzzing.

The results of performing code review on static analysis results, gives us a list of unsafe instances with either exploitability as HIGH or LOW. We may now consider exploiting those unsafe instances marked as HIGH by running the vulnerable system under test with random inputs which could be either valid or invalid inputs.

**Fuzzing**
The unsafe instances that are found after static analysis and code review, by our approach, are marked either HIGH or LOW, as their exploitability. Then we retrieved an unsafe instance from the Vulnerability and Threat database with HIGH tag. Now we identified that unsafe instance in the code, and guarded that, just before and after, by adding debugging statements. Then we run fuzzing (see Figure 4.4).
Figure 4.4: Fuzzing iteratively to ensure security testing.

After fuzzing, the debugger to which the SUT is attached primarily, would generate debug logs. We use the debug logs along with summary report, collector and the program output in order to analyze the results. We do this step to find if the identified unsafe instance with HIGH exploitability was exploited in real. Signs of exploitation are buffer overflow that result into segmentation fault, exception that may result in program abrupt termination etc. Then we update the unsafe instance to the Vulnerability and Threat Database as CONFIRMED. Then we identified that a range check can be a proper fix for this vulnerability. Then we applied a range check countermeasure on the unsafe instance. In another case, we replaced the unsafe instance, which was a library function, with another safer library function accompanied by a range check on it.

After applying the countermeasure on the unsafe instance, we update the Vulnerability and Threat Database. Update is made on the basis that a countermeasure has already been applied on the unsafe instance. We update this event by marking the unsafe instance as APPLIED. After this, we need to compile the SUT because we have modified in the code. In the next iteration of fuzzing, which we do after the first iteration, if the same exploitation that occurred on to the previously investigated instance do not appear again, then we can update it as CHECKED in the database. The other update that could be made is insertion of the newly found unknown vulnerabilities to the Vulnerability and Threat Database that were found in the process of repeated fuzzing. We have seen it happening in our experiments.

We propose to repeat the fuzzing process, as we have discussed already that we do it, because we want to confirm ourselves that the countermeasure we apply, indeed works. Fuzzing for the first time only allows us to know whether the unsafe instance was exploited. Then we correspondingly apply countermeasure on it. The next fuzzing iteration would then help us to check whether our countermeasure works. If not, then we need to find a countermeasure by checking
through error propagation technique that we used and demonstrated in section 5.3 (Observation III). Once we have applied the countermeasure again, we need to confirm the same by fuzzing. So we run fuzzing again and again until the vulnerability is completely gone, which we do in our experiment. We have used the term “CHECKED” to update the database for the event when we are completely sure that our countermeasure has worked. In our results in chapter 5, we show the benefits of using repeated fuzzing in our security evaluation framework.

We propose to repeat fuzzing to all of the unsafe instances that we marked with “exploitability” as “HIGH”, just after they are CONFIRMED as exploited. After countermeasure has been applied to all of those unsafe instances, the corresponding unsafe instances should be marked as APPLIED in the Vulnerability and Threat Database and then fuzzing should be run again to check if the countermeasure is successful.
5 Experimental Results and Evaluation

In this chapter, we shall discuss the technical setup that is required for conducting experiments with our security evaluation framework. We also present the results of the conducted experiments that were carried out in order to evaluate security in automotive software. The evaluation comprises detailed analysis of the results and observations. In Table A.3, we show the benefits of performing code review on the results of static analysis. In Table A.9 and Figure 5.5 we show the benefits of repetitive fuzzing. We also discuss the additional benefits that we found with repetitive fuzzing in Appendix B, section B.3. In section B.1, we explain our methodology of analysis of results with examples and in section B.2 we also elaborate our method that we generalized to be used for choosing and applying countermeasures within this framework. Finally, in Table 5.1, we compare the advantages and disadvantages of using the security testing methods in different order of combinations. This table clarifies the definitive advantages that we derived from Table 2.1 and the results of investigation that we conducted are the different combinations. Among all combinations we just chose one that gives us the maximum benefits.
5.1 Experimental Setup

5.1.1 Minimum PC Requirements

| RAM: 768 MB |
| HDD: 8 GB |

5.1.2 Software Requirements

Virtualization software used is Oracle VM Virtual Box 4.3.28 r98988
Operating system used is Ubuntu Linux 14.04 LTS
Compiler: GCC (C and C++) available in Linux distribution
Tools used for static analysis are Rats, Flawfinder and Klocwork.
“fuzzer.c” – Copyright (c) 1989 Lars Fredriksen, Bryan So, Barton Miller

5.1.3 Software Usage

**Flawfinder:**
flawfinder filename.c

**Rats:**
rats –w 1 filename.c HIGH
rats –w 2 filename.c MEDIUM
rats –w 3 filename.c LOW

5.1.4 System Under Test (SUT) Architecture

Here we present the architecture of the software, known as the SUT in this thesis. The uppermost layer consists of the software components. As mentioned in chapter 3, a software component (SWC) is a set of C files and each software component has its own specific functionality. Components like Gateway, PTO, and Vehicle Speed have been used as a part of the SUT. The next layer is the Run Time Environment (RTE) layer that handles the interaction between SWC and Operating System (OS) and hardware (H/W). The OS layer contains system level functionalities like util, CAN and IO. Lastly, the SUT gets the input from the hardware in the main function.
5.1.5 Scope

We have performed security testing at the software component level of abstraction which lie in the middle of the three abstraction layers of the E/E systems architecture. We perform white box testing at this level of abstraction.

**Figure 5.2:** Scope of security testing.
5.1.6 Fuzzing setup

In this section, we present the experimental setup for fuzzing which includes adding a debugger to SUT, generating the default call graph of an intended result, generating test cases, and finally we elaborate the dispatcher and assessor setup.

Add debugger

Before running our experiment, a debugger is needed to be attached to the software to generate the control flow path of the runtime execution of the SUT. Control flow statements are required to generate the call graph sequence of the functions during the execution flow of the software. We use “debug macros” that works very much like “printf” statements. Usually “debug macros” embed “fprintf” within itself could help the programmer to generate code based on his choice, for example, info, debug, warn or error. As a result a predefined template of such message constructs would save time for programmer instead of writing “printf” statements every time. We use only logs of type debug to identify the control flow of the SUT. These debugging statements would be useful, if there was a vulnerability in the code that could lead to the termination of software execution, then the security testers would like to know until which function the program executed and where it was terminated. Then, the occurrence of the software bug or the security flaw could be easily identified in the code. The debugger statements are also required to confirm that those unsafe instances that were identified during static analysis followed by code review, would really overflow or not during fuzzing.

Security Tester’s role

- Add debug.h to SUT
- Include debug.h to every debug statement for every function.
- Compile and run the SUT with valid and correct inputs.
- Generate a call graph.
To include debugging statements in the software code, a debugger should be included in the framework as the initial setup before fuzzing begins (see Figure 5.3). The security tester must include the “debug.h” in the workspace of the code. This debugger would enable debugging in the user code. The security tester can inject the debug statements before and after those unsafe instances in the code. After the security tester includes the debugging statements in the software, and the necessary inclusion of the “debug.h” in the associated header files, the entire workspace must be compiled.

**Generate call graph**

To validate that the execution path found in the call graph could be faulty, we initially generated a sample debug output of the correct execution path. Then we compared the faulty execution paths against this correct execution path. A valid and correct input (see Figure 5.3) is one that produces the intended result. The security tester should generate some correct input values that are believed to be correct and should produce the intended result. Then the security tester must dispatch this input to the SUT and run the SUT. After this step, the call graph of a correct execution would be generated.

**Generate test cases**

Goal:

- Generate wide variety of valid and invalid random test-cases to connect with the SUT for execution.
Different input types:
- Valid
- Invalid

Valid Input
**Generate** – We generate valid inputs of lengths that are in the acceptable range that do not allow the parsers to create immediate problems to the program execution. An example could be use of a buffer size lower than the actual limit. If the allowed length of a frame should be 16 characters, we can generate frames of sizes 8, 10, 12, 14, or 16 characters.

**Mutate** – We mutate the valid input in such a way that it does not cause immediate problems. For example, adding a negative character before the input could allow to execute the SUT, at the same time could stress it.

Invalid Input
**Generate** – We generate inputs of size far greater than the allowed range or negative size full with garbage values.

**Mutate** – We add negative sign before the invalid input. Then we performed manual fuzzing with the mutated valid inputs.

Intelligent fuzzing
We perform fuzzing intelligently by taking the valid and invalid input after generation and then mutating that in an intelligent fashion. In the experiment that we conducted, we needed two arguments for input to the main function. We used two arguments in such a way, that if only one type (valid/invalid) is used, then the same input value of that type can be used for both the arguments (see Appendix A, Table A.4). Next, we observed the behavior of the arguments i.e. the value that is accepted by the SUT against the value that has been supplied (see the 2nd row of Table A.4). We then manipulated the input by adding negative sign before the first input and changed the second input to a different one (see Table A.8). We minutely observed whether an error value would crash the SUT at some point or run safely till the end. There is also the third possibility that the error value would not cause any problem to majority of the statements and functions, but that
value may be vulnerable to some functions causing failure value in the output, which we also found as true.

**Test Case Generator**

We generated the valid and invalid test cases with the invocation of a bash script, known as *Test Case Generator* (see Figure 5.4), that automatically generated valid test cases of 8, 10 and 16 character lengths and invalid test cases of any random length. For extra level of security, we generated these test cases in the binary format, which were directly stored in the testbed in binary.

**Dispatcher**

*Dispatcher* (see Figure 5.4) is a functionality in the framework that helps to connect the inputs to the SUT. It is a bash script that we use to request fuzzing. Without having prior knowledge of the fuzzing process the security tester should be able to use the dispatcher. The dispatcher connects the SUT’s binary with two inputs as its two arguments and generates debug log and program output.

**Assessor**

The security tester should observe the results from fuzzing, by watching the assessor. The assessor is a functionality in this framework (see Figure 5.4) that consists of the vulnerable SUT’s output and a debug log report. The output file contains the print statements that would help the security tester to analyze the result values if that contains any failure value.

On the other hand, the debug log consists of the execution sequence of the debug statements in the SUT. This debug log file would help us to identify the exact place where in the SUT the crash-level software bug exist that pose as a threat to exploit. There are also other results from where we can get more information. These are the test cases that: hanged, timed out, had segmentation fault, had exception and no failures. We collect all of these test cases in a repository known as the *Collector* (see Figure 5.4) of the framework. To aid the study of the results from collector, we also generate a summary report that gives total count of the respective faults as its output. The test cases that are at fault would be translated back to textual format from the binary format. We added this functionality because we needed to look into the test cases at fault for further investigation.
In order to test that repetitive fuzzing indeed works in this security evaluation framework, we have set up the fuzzing system in a way that the inherent operations of fuzzing are fully automated, and can be used as a black box to a normal user. It works in the following way:

a. First we require to set up our test bed. As discussed earlier, we invoke our Test Case Generator, with a specified size (8 or 10 or 16 as valid and without specified size gives invalid test cases) of the target CAN Frame and total required number of test cases, as its input arguments. The result is that in the testbed, we will get automatically generated test cases with the desired size and as many test cases requested.

b. Then, to start fuzzing, we need to connect these test cases (from the test bed) to the SUT at run time. We do that by invoking our fully automated Dispatcher, which is also a bash
script, provided with the test cases range that we want to use as inputs in fuzzing operation. The Dispatcher in turn runs the SUT continuously while connecting the test cases as inputs one after another and displaying corresponding result to the terminal automatically. The results of the Dispatcher operation for fuzzing operation of every input are piped into Assessor, Collector and Summary Report. The Assessor comprises the debug logs and program output of the fuzzing operation. The Collector collects the test cases back depending upon their execution result and are piped to the following storage spaces in the collector:

1. Hangs – contains the test cases for which the SUT hanged.
2. Time out – contains the test cases for which the SUT timed out.
3. Segmentation fault – contains the test cases for which the SUT crashed due to buffer overflow resulting into segmentation fault.
4. Exception - contains the test cases for which the SUT crashed due to unhandled exception.
5. No failures - contains the test cases for which the SUT had no failures.

The Summary Report is a report that will contain the total count of the individual result types (mentioned above) in the fuzzing operation.

c. To repeat fuzzing, no special change is required, except re-compiling the SUT after the code in the SUT has been changed due to applying debugging statements and/or applying countermeasures.

5.2 Experimental Results
In this section, we discuss the benefits of the methods in our security evaluation framework that we observed while carrying our security evaluation experiments in the automotive software.

5.2.1 Results of Static analysis
From the results of static analysis (see Table A.2), we can see that we identified a specific pattern, the potential vulnerability and kingdom number of the unsafe instance, by using our own method
of vulnerability analysis, as discussed in chapter 4 (section 4.1). Then we have identified the potential threat. The threat identification method is based on the assumption and knowledge of security testing. Though we primarily produce our assumption of the threat, in reality we do not rely on it. We assume that it can be any of the threats that are present in the STRIDE model. In fuzzing, when we get a clear picture of the kind of the exploitation that occurred on those instances, we formally decide on the threat that happened and therefore update the same in the vulnerability and threat database.

5.2.2 Results of Code review
We do code review on the results of static analysis that we produced in the previous step. We obtain the exploitability of the unsafe instances by manual investigation in the code. In the results of static analysis we saw that the static analysis tools, *Klockwork*, for example, produced five unsafe instances as result (see Table A.2). By doing code review, we found only two results are meaningful and we rate the exploitability of those two instances are HIGH, the other three as LOW. We found that these two unsafe instances, which we rate as HIGH, may be overflowed because they are taking user input in their parameter. Thus we see, the search space is already reduced to 60%. In the next step, i.e. fuzzing we will consider only these two unsafe instances for investigation.

5.2.3 Result of pre - Fuzzing with correct input
Before the start of fuzzing, we need just one-time setup to generate a call graph (see Figure A.1) that would represent an intended execution of the SUT that we are supposed to test. This call graph is the output of this stage that is generated after running the SUT with correct and valid inputs.

5.2.4 Results of Fuzzing
As mentioned earlier, we do fuzzing on the results of code review, more specifically, on the unsafe instances that we rated with exploitability as HIGH. By fuzzing on these unsafe instances for the first time, we get the result of the exploitation on these unsafe instances as a result. If we observe an exploitation, then we do the following three things:

a. We first update the vulnerability and threat database with the threat that we observed after exploitation. For example, if in static analysis, we assumed that the threat on the instance
“atoi ()” would be Denial of Service (DOS), in fuzzing we found that there was no DOS but “Tampering of Data” (see Table A.9), then we update it. Then we mark that instance as CONFIRMED.

b. In the next step, we apply a countermeasure on that instance in the SUT. Then we update the unsafe instance as APPLIED in the vulnerability and threat database.

c. In the next step, we compile the SUT and repeat fuzzing. The benefit of repetitive fuzzing is to confirm the result of our countermeasure, which we applied after the previous iteration of fuzzing, whether really worked or not. If that worked, we need to update the unsafe instance as CHECKED.

Apart from these benefits, we also observed that during fuzzing, we are able to find more unsafe instances in the code that can pose as a vulnerability. We call these vulnerabilities as software bugs that may cause problems in code. For example, by using repetitive fuzzing, we found that the exploitation on the unsafe instance [memcpy(), swc_vehicle_speed.c, 52] occurred due to an unknown vulnerability present in the function [CalcSpeedFromTachoData(), swc_vehicle_speed.c] (see Table A.9). When we found that applying a countermeasure on the “memcpy” instance did not work, we investigated the instance with the help of error propagation technique and found that the instance CalcSpeedFromTachoData() was responsible. We immediately inserted this new instance to the vulnerability and threat database, then performed vulnerability analysis, threat analysis and code review on it. Then we applied a countermeasure on it and repeated fuzzing. Then in result we found that we have prevented the exploitation on the “memcpy” instance. Using our security evaluation framework, we ran fuzzing with 1000 valid and 1000 invalid test cases. By fuzzing with 1000 valid test cases (see Figure 5.5), in the first iteration, we found 935 test cases at fault (see left of Figure 5.5) due to one unsafe instance causing unhandled exception in the code. This fault caused termination of the SUT. We found 65 test cases with no failure. After applying countermeasures by using our approach of finding and applying countermeasure on the identified vulnerability, we were able to prevent the system crash of the SUT. After few iterations (see right of Figure 5.5) we reduced the fault to 0.
Valid test cases 16 characters long

Figure 5.5: Results of fuzzing with valid-16 test cases before and after applying countermeasures.

In another example, we ran fuzzing on 1000 invalid test cases (see Figure 5.6). This experiment found segmentation fault due to buffer overflow on the “memcpy” instance that we discussed before. In this case, we found 972 test cases at segmentation faults, 3 test cases at exception and 25 no failure (see left of Figure 5.6). We observed that all of the system crashes in SUT related to segmentation fault occurred due to just one vulnerability, on which we finally applied countermeasure. Similarly, all of the 25 exception were caused due to the same unhandled exception vulnerability, on which we also applied countermeasure. Then we do another iteration of fuzzing. We can observe the result (see right of Figure 5.6) that we were successful to reduce all the faults to 0 by using our approach.

Invalid test cases of random length

Figure 5.6: Results of fuzzing with invalid test cases before and after applying countermeasures.
In total, we ran 11800 test cases for the first fuzzing iteration and used the same test cases for the next iterations after applying the countermeasures. From the experiments, we found that mainly three types of faults are crashing the SUT, which are segmentation faults, exceptions and hangs. A segmentation fault can affect the ECU to a great extent. In some cases, it results into incorrect memory reads inside the ECU. However, other OS’s coupled with hardware that has memory R/W protection would cause a trap and then it depends on the handler what is needed to be done next.

In the automotive domain, the ideal strategy to handle such scenario is to log the fault and restart the ECU. For exceptions, the strategy would be the same. In case of hangs, there are watchdog timers that catches the timeout, logs the faults and restarts the ECU.

We also found that tampering of data has been happening with the supplied input that modifies the actual data resulting into flow of wrong values. It stays silent throughout the course of the program flow. Strategies to handle such silent wrong values would be to use multiple redundant sources of information in case it is safety critical. In that case, another signal source is used if one is deemed faulty. Another strategy is to disable functionality and warn the driver about it, or perhaps to enter a limp home mode where safest possible parameters are used given the situation is detected.

While modifying the code during countermeasure implementation, we found that the value of the speed parameter has been fluctuating hugely towards large numbers and also sometimes towards negativity. Invalidated value of vehicle speed would affect a great many things such as adaptive cruise control, speed limiter, entry conditions of several diagnostic routines etc. It may happen that the developer never thought of large values or negative values that may harm the system under test, and therefore the SUT lacked invocation of safety critical functions under such situation. For example, some services that are intended to be disabled in order to switch to safety mode, would not be disabled, resulting into serious internal activity occurring inside the ECU, such as abrupt restart of ECU or displaying wrong values on the speed limiter, as for example.

5.3 Discussion

In this section we will discuss about the observations and crucial findings from this thesis work.
Observation I

MISRA C Violation

Divide by Zero in the code had been identified using dynamic analysis technique, fuzzing. A problem occurred when there was a ‘/’ division operator and without saving the denominator in a variable and without checking for its range validation, the division was performed. This violates the MISRA C rule 21.1 which states that run-time failures should be minimized by static analysis, dynamic analysis or explicit code checking.

By “minimized”, it means programmers should be aware of common programmatic errors and avoid such mistakes during coding, which are not explicitly mentioned in MISRA C guidelines. In the code, it is found that the range check was not made for the denominator, in a division operation, which violated the rule “a run-time check may not be necessary, provided it can be demonstrated that for all values within the defined range the exception cannot occur. Such a demonstration, if used, should be documented along with the assumptions on which it depends.”[60]

Observation II

Data Pipes

It has been observed in the source code that the input is used directly to the code after being received from the user. The input is carried by and propagated through few functions in the source code rather than the major parts of the code. We can derive from here, that there are few data pipes through which the input data flows to some parts of the entire source code (see Figure 5.7). Rest of the code, as seen in the diagram (Figure 5.7) could stay unaffected by the input. Any of these data pipes may contain one of these identified unsafe instances if at least one of the arguments to this unsafe instance is getting the value from the input data.

In the following diagram (see Figure 5.7), we can observe that the user input is first taken at the beginning of the program execution after which it is stored into an output array.
The particular element “R” responsible for failure of the unsafe instance “memcpy” is a value of a particular position of the output array that represent the value of the “speedLength” variable in the representative code. In the previous part of the code, before this unsafe instance “memcpy”, this element “R” was not used at all in some functions which are represented by dots in the above diagram. Then at some point, the region represented by dashes in the above diagram, it was observed by backtracking “R” from “memcpy” that “R” was extracted using a function and stored in a variable and it kept being used by the following code until it reached the “memcpy” unsafe instance. We will again see this in detail later in this report.

So, it could be inferred that the input data are flowing through data pipes as shown in the diagram above. And from the static analysis and code review, it could be revealed that some of these unsafe instances are actually a stage in this data pipes. Therefore, investigating only these
data pipes could help us identify the reason of the software being crashed due to the bad inputs which could confirm the vulnerability at the end.

Observation III

Error Propagation

Propagation of error has been found valid in our case when we run the automotive code on our security evaluation framework. By performing code review on the identified unsafe instance, we found, in fact, error has propagated from the place where the input is taken and processed until some function calls afterwards in the code. We also found that this error finally terminated the program when it encountered one of the unsafe instances reported by the static analysis method in our framework. We claim, by listing the order of the functions (see Figure 5.8), through which the error traversed, that we can choose to place our debug statements (see Figure 5.9). After running fuzzing, we can decide to choose the location of the countermeasure. The following example will explain the process that we claimed here in this section.

![Diagram of error propagation]

**Figure 5.8:** Error Propagation Scenario of segmentation fault.
In this example, the unsafe instance, a library “memcpy” function is traced in the source code of the software. A software security tester investigates and finds that the parameter R passed as an argument to “memcpy”, could be responsible for a buffer overflow on the destination buffer m1.

**Figure 5.9**: Countermeasure adoption and error value propagation.

```c
memcpy(outputData, &inputData[6], speedLength);
```
Tracing backwards in the code, we find that this responsible variable “R” was used as an argument to the function “calcSpeedFromTachoData()” and stored in a speed variable.

```c
speed = CalcSpeedFromTachoData(&inputData[6], speedLength, &inputData[0]);
```

Then a line backwards, we find that the responsible variable “R” that corresponds to “speedLength” here, got the assignment value from an array at fifth element position which is “inputData[5]”.

```c
CAN_GetFrame(CAN_TACHO_DATA, inputData);
speedLength = inputData[5];
```

The function “CAN_GetFrame” gets the value of the recent frame in the “inputData” array.

So, now we have all those functions, through which the error in the form of the responsible element “R” propagated. Now we choose (see Figure 5.9) in which functions range checks should be incorporated to prevent the flow of the error that was introduced at the input stage.

**Observation IV**

**Unknown Vulnerability Exposure**

By using this security evaluation framework, we have observed that vulnerabilities of unknown types would be exposed. The unknown vulnerabilities can originate from programming mistakes from the implementation phase. One example is unhandled run-time exception that usually happen in code without following international coding standard, such as MISRA-C.

In this observation, we found that other unknown vulnerabilities could be found few lines backward to the identified unsafe instance. Such unknown vulnerabilities would be confirmed by adding debugging comments and running fuzzing.
Observation V

**Table 5.1: Comparison of combined methodologies in security testing.**

<table>
<thead>
<tr>
<th>Method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| Static analysis + Code review | - Static analysis gives result on which we do code review. By doing so,  
   - We are getting a good starting point of security testing.  
   - Found unsafe instances are checked for exploitability.  
   - Search space for finding vulnerabilities is minimized.  
   - Unknown vulnerabilities may be found immediately adjacent to (before) the found unsafe instance. | - All kind of vulnerabilities would not be considered because the method lacks input to SUT.  
- Input to the system under test may give rise to runtime vulnerabilities which is missing in this approach. |
| Static analysis + fuzzing     | - Only the known instances from static analysis, that are overflowed could be confirmed. | - Practically the method could cost very high depending upon the number of lines of code.  
- If the number of unsafe instances are in the order of thousands, then debugging on those many unsafe instances would be a huge task.  
- It may be termed as blind fold fuzzing because we won’t uncover the unknown vulnerabilities. |
| Static analysis + repetitive (Code review + fuzzing) | - Helps us to find the appropriate places for mitigation by uncovering unprecedented software bugs through repetitive fuzzing accompanied by error propagation identification technique in the program that could possibly catch and handle vulnerabilities at correct places. | - Overhead of a database to track vulnerabilities as checklist. |
| (repetitive) (Code review + fuzzing) | - Uncovers dormant vulnerabilities that can cause major problems in the later stages, if not properly addressed for mitigation. E.g. negative speed, impossibly large values for speed. | - Iteration needed as long as results do not report any vulnerability.  
- A debugger is needed. |
Observation VI

Feasibility study of the security testing methods

**Table 5.2:** Feasibility study of different security testing methods at different levels of abstraction.

<table>
<thead>
<tr>
<th>Method</th>
<th>Software Unit</th>
<th>Software Component</th>
<th>ECU Software</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Black box</strong></td>
<td>Fuzzing</td>
<td>Fuzzing</td>
<td>Fuzzing</td>
</tr>
<tr>
<td><strong>White box</strong></td>
<td>Static analysis, Code review, Fuzzing</td>
<td>Static analysis, Code review, Fuzzing</td>
<td>Static analysis, Code review, Fuzzing</td>
</tr>
</tbody>
</table>

As we see from the table above (see Table 5.2) that fuzzing could be done as either on black box or white box at all the levels of abstraction be it a software unit, component or the whole ECU software. On the other hand, static analysis which is strictly dependent on the source code could only be run on white box at all the levels of abstraction of automotive software.

Lastly, code review, a thoroughly manual process, is also done directly on the source code. Therefore it should be done only in a white box manner. On the lowest levels of abstraction, it is possible to do manual code review. But at the highest level, if an ECU software contains millions of lines of code then manual code review on all of the lines of code is impossible.

In this framework, code review is done on the results of static analysis, which are unsafe instances. Those unsafe instances are investigated in the code review process which limited down the search space area with associated definitive advantages, such catching hold of the data pipes through which input data flows, reduced required iterations of fuzzing and reduced overall time for security testing.

At the end, fuzzing on the results of code review allowed maximum code coverage by which we can definitely point out that we have uncovered the data flow paths that code review gave us as a hint. Fuzzing with the help of huge number of valid and invalid test cases ensures us the correct compatibility of the synchronized data flow paths with code review. As many as test cases we used, we uncovered more data flow paths, that proves our assumption as valid and validates the correct combination of methods to be used.
6 Conclusions

From the experiments, we found that by performing vulnerability analysis during static analysis, we can assimilate the knowledge of potential vulnerability and threats found during this method to accompany further investigation during code review. Again, we can use this knowledge to choose and apply different countermeasures, on account of the vulnerabilities that have been exposed during fuzzing.

We also found that injecting code review after static analysis, followed by vulnerability analysis, the search space of finding vulnerabilities in code would be reduced radically. Then we found that fuzzing is able to identify major sources of the buffer overflows that could lead to any kind of the attacks described in the STRIDE model. Fuzzing is also capable of uncovering unintentional programming errors that causes exception at runtime, leading to Denial of Service, which static analysis is not capable of doing. e.g., unhandled divide by zero exception.

Fuzzing together with static analysis and code review would also identify known and unknown vulnerabilities in code through repetitive fuzzing and error propagation identification technique. Fuzzing alone helps us to identify another source of dormant vulnerability known as failure value that could lead to the tampering of data or any other kinds of attacks. e.g., unusually large positive numbers could turn out to be a negative or different positive number that could lead to confusion and wrong result.

Future Work

The framework introduced in the earlier chapters, discussed about how security testing could be confirmed and how countermeasures could be incorporated to prevent the vulnerabilities and threats to the system under test. Though the framework is standing in an efficient manner by now with the combination of the security testing methods, yet most of the work and knowledge of the system is required by the security tester prior to performing security testing using this framework. Automating the generation of the Vulnerability & Threat Database, requesting unsafe instance from the Vulnerability & Threat Database, applying automatically to SUT and automatically sending compile request to the SUT are some of the technical functionalities that are missing from the work which if implemented could save some time and effort for the security tester.
References


[56] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, “Comprehensive Experimental Analyses of Automotive Attack Surfaces”.


# Appendix A

Table A.1: HEAVENS Terminologies [12] on Security

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Establish logical or physical communication or otherwise interaction with assets and/or TOEs.</td>
</tr>
<tr>
<td>Access Control</td>
<td>Means to ensure access rights for entities in relation to assets.</td>
</tr>
<tr>
<td>Adverse Action</td>
<td>Adverse actions are the actions that are performed by attackers to mount attacks on an asset and/or a TOE.</td>
</tr>
<tr>
<td>Asset</td>
<td>Anything that has a value to any of the stakeholders, that could potentially be subject to attacks and possibly, but not necessarily, motivates countermeasures.</td>
</tr>
<tr>
<td>Attack</td>
<td>A series of adverse actions performed by attackers to achieve unauthorized results in relation to assets and/or TOEs.</td>
</tr>
<tr>
<td>Attacker</td>
<td>Any type of individual, group or entity aiming to mount attacks.</td>
</tr>
<tr>
<td>Attack Vector</td>
<td>An attack vector is a set of adverse actions to establish a path or means to mount an attack.</td>
</tr>
<tr>
<td>Authentication</td>
<td>Authentication is the process of assuring and verifying authenticity.</td>
</tr>
<tr>
<td>Authenticity</td>
<td>Authenticity is an attribute to establish that an entity is what it claims to be.</td>
</tr>
<tr>
<td>Authorization</td>
<td>Authorization is an attribute that enforces access control.</td>
</tr>
<tr>
<td>Availability</td>
<td>Availability is an attribute that ensures correct and timely access upon demand by an authorized entity.</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Confidentiality is an attribute to ensure that information is not disclosed to an unauthorized entity.</td>
</tr>
<tr>
<td>Countermeasure</td>
<td>Technical solution depending on the estimated security level of an asset and/or a TOE in order to avoid unreasonable risk and achieve or maintain a secure state.</td>
</tr>
<tr>
<td>Integrity</td>
<td>Integrity is an attribute that protects the accuracy and completeness of assets by preventing unauthorized modifications.</td>
</tr>
<tr>
<td>Severity</td>
<td>Estimate of the magnitude of harm to stakeholders originating from an attack.</td>
</tr>
<tr>
<td>Target Of Evaluation( TOE)</td>
<td>A TOE is a set of assets.</td>
</tr>
<tr>
<td>Threat</td>
<td>Potential cause of an unwanted incident, which may result in harm to a system or organization.</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Vulnerability</td>
<td>Weakness in an asset and/or a TOE that can potentially be exploited by attackers to mount attacks.</td>
</tr>
</tbody>
</table>

Table A.2: Vulnerability and threat database

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Flaw finder</td>
<td>atoi, main.c, 30</td>
<td>Unless checked the resulting number can exceed the expected range. Large numbers can roll over into negative numbers.</td>
<td>Integer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Denial of service or Tampering of data</td>
</tr>
<tr>
<td></td>
<td>memcpy, can.c, 15</td>
<td>Does not check for buffer overflows when copying to destination.</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td></td>
<td>memcpy, can.c, 24</td>
<td>Does not check for buffer overflows when copying to destination.</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Flaw finder</td>
<td>memcpy, swc_vehicle_speed.c, 52</td>
<td>Does not check for buffer overflows when copying to destination.</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------</td>
<td>---------------------------------------------------------------</td>
<td>-----------------</td>
<td>------------------------</td>
<td>---</td>
<td>------------------------</td>
</tr>
<tr>
<td>Flaw finder</td>
<td>strlen, util.c, 27</td>
<td>Does not handle strings that are not \0-terminated.</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Denial of service</td>
</tr>
<tr>
<td>Rats</td>
<td>memcpy, can.c, 15</td>
<td>Does not check for buffer overflows when copying to destination</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Rats</td>
<td>memcpy, can.c, 24</td>
<td>Does not check for buffer overflows when copying to destination</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Rats</td>
<td>memcpy, swc_vehicle_speed.c, 52</td>
<td>Does not check for buffer overflows when copying to destination</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Source</td>
<td>File</td>
<td>Function</td>
<td>Description</td>
<td>Severity</td>
<td>Vulnerability</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td>----------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td>Rats</td>
<td>strlen, util.c, 27</td>
<td>This function does not properly handle non-NULL terminated strings. This does not result in exploitable code but can lead to access violations</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Denial of service</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>atoi, main.c, 30</td>
<td>Function atoi is deprecated</td>
<td>Integer overflow</td>
<td>1</td>
<td>Denial of service or Tampering of data</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>memcpy, can.c, 15</td>
<td>Function memcpy is deprecated</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>memcpy, can.c, 24</td>
<td>Function memcpy is deprecated</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>memcpy, swc_vehicle_speed.c, 52</td>
<td>Function memcpy is deprecated</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>strlen, util.c, 27</td>
<td>Function strlen is deprecated</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Denial of service</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>------------------</td>
<td>------------------------------</td>
<td>--------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>Flaw finder</td>
<td>atoi, main.c, 30</td>
<td>Unless checked the resulting number can exceed the expected range. Large numbers can roll over into negative numbers.</td>
<td>Integer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Denial of service or Tampering of data</td>
</tr>
<tr>
<td>Flaw finder</td>
<td>memcpy, can.c, 15</td>
<td>Does not check for buffer overflows when copying to destination.</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Flaw finder</td>
<td>memcpy, can.c, 24</td>
<td>Does not check for buffer overflows when copying to destination.</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Flaw finder</td>
<td>memcpy, swc_veh_icle_speed.c, 52</td>
<td>Does not check for buffer overflows when copying to destination.</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Flaw finder</td>
<td>strlen, util.c, 27</td>
<td>Does not handle strings that are not \0-terminated.</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Denial of service</td>
</tr>
<tr>
<td>Rats</td>
<td>memcpy, can.c, 15</td>
<td>Does not check for buffer overflows when copying to destination.</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Rats</td>
<td>memcpy, can.c, 24</td>
<td>Does not check for buffer overflows when</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Software</td>
<td>Function</td>
<td>File</td>
<td>Source Code Line</td>
<td>Description</td>
<td>Impact</td>
<td>Severity</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>------</td>
<td>------------------</td>
<td>-------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>Rats</td>
<td>memcpy</td>
<td>swc_vehicle_speed.c, 52</td>
<td>Does not check for buffer overflows when copying to destination</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Rats</td>
<td>strlen</td>
<td>util.c, 27</td>
<td>This function does not properly handle non-NULL terminated strings. This does not result in exploitable code but can lead to access violations</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Denial of service</td>
</tr>
<tr>
<td>Klocwork</td>
<td>atoi</td>
<td>main.c, 30</td>
<td>Function atoi is deprecated</td>
<td>Integer overflow</td>
<td>1</td>
<td>Denial of service or Tampering of data</td>
</tr>
<tr>
<td>Klocwork</td>
<td>memcpy</td>
<td>can.c, 15</td>
<td>Function memcpy is deprecated</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Klocwork</td>
<td>memcpy</td>
<td>can.c, 24</td>
<td>Function memcpy is deprecated</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Klocwork</td>
<td>memcpy</td>
<td>swc_vehicle_speed.c, 52</td>
<td>Function memcpy is deprecated</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
</tr>
<tr>
<td>Klocwork</td>
<td>strlen</td>
<td>util.c, 27</td>
<td>Function strlen is deprecated</td>
<td>Buffer overflow</td>
<td>1</td>
<td>Denial of service</td>
</tr>
</tbody>
</table>
Figure A.1: Call Graph of a correct execution of the SUT

Note: Every ‘X’ represent a separate character (valid for Tables A.4 – A.6)

Table A.4: Valid input arguments 8 characters long

<table>
<thead>
<tr>
<th></th>
<th>Arg1 Valid 8 characters</th>
<th>Arg2 Valid 8 characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual input</td>
<td>XXXXXXXXXX</td>
<td>XXXXXXXXXX</td>
</tr>
<tr>
<td>Actual behavior</td>
<td>XXXXXXXXXX</td>
<td>0</td>
</tr>
</tbody>
</table>

Table A.5: Valid input arguments 10 characters long

<table>
<thead>
<tr>
<th></th>
<th>Arg1 Valid 10 characters</th>
<th>Arg2 Valid 10 characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual input</td>
<td>XXXXXXXXXXXX</td>
<td>XXXXXXXXXXXX</td>
</tr>
<tr>
<td>Actual behavior</td>
<td>XXXXXXXXXXXX</td>
<td>0</td>
</tr>
</tbody>
</table>

Table A.6: Valid input arguments 16 characters long

<table>
<thead>
<tr>
<th></th>
<th>Arg1 Valid 16 characters</th>
<th>Arg2 Valid 16 characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual input</td>
<td>XXXXXXXXXXXXXXXXXXXXXX</td>
<td>XXXXXXXXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>Actual behavior</td>
<td>XXXXXXXXXXXXXXXXXXXXXX</td>
<td>0</td>
</tr>
</tbody>
</table>
Note: Every ‘X’ in X...n represent a separate character, Y represent a string of different pattern than the actual input (valid for table A.7 – A.8)

**Table A.7:** Invalid input arguments of random length

<table>
<thead>
<tr>
<th></th>
<th>Arg1 Invalid random length</th>
<th>Arg2 Invalid random length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual input</td>
<td>X..............................n</td>
<td>X..............................n</td>
</tr>
<tr>
<td>Actual behavior</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table A.8:** Invalid arguments with negative characters

<table>
<thead>
<tr>
<th></th>
<th>Arg1 Invalid random length</th>
<th>Arg2 valid random length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual input</td>
<td>-X.............................n</td>
<td>0&lt;n&lt;65535</td>
</tr>
<tr>
<td>Actual behavior</td>
<td>Y</td>
<td>0 – 65535</td>
</tr>
</tbody>
</table>

**Table A.9:** Updated Vulnerability & Threat Database after fuzzing

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>atoi, main.c, 30</td>
<td>Unless checked the resulting number exceeded the expected range. Large numbers rolled over into negative numbers.</td>
<td>Integer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Denial of service</td>
<td>HIGH</td>
<td>The size of the source depended indirectly on the input from the user that overflowed the destination buffer at runtime. NO DOS however value failure observed</td>
</tr>
<tr>
<td>memcpy, can.c, 15</td>
<td>Does not check for buffer overflow</td>
<td>Buffer overflow</td>
<td>Buffer [BUFF] Overflow</td>
<td>1</td>
<td>Elevation of privilege</td>
<td>LOW</td>
<td>Ignored</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Error Type</td>
<td>Severity</td>
<td>Status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------</td>
<td>----------</td>
<td>------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcpy, can.c, 24</td>
<td>Does not check for buffer overflow when copying to destination.</td>
<td>Buffer overflow</td>
<td>LOW</td>
<td>Ignored</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcpy, swc_vehicle_speed.c, 52</td>
<td>Did not check for buffer overflow when copying to destination.</td>
<td>Buffer overflow</td>
<td>HIGH</td>
<td>CONFIRMED APPLIED CHECKED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strlen, util.c, 27</td>
<td>Does not handle strings that are not \0-terminated.</td>
<td>Buffer overflow</td>
<td>LOW</td>
<td>Ignored</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CalcSpeedFromTachoData, swc_veh</td>
<td>Integer Divide by zero Floating</td>
<td>Software bug</td>
<td>HIGH</td>
<td>CONFIRMED APPLIED CHECKED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table A-10: Asset range values to be used in countermeasure

<table>
<thead>
<tr>
<th>Asset</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>speedLength</td>
<td>0 – 8</td>
</tr>
<tr>
<td>Speed</td>
<td>0 – 200</td>
</tr>
</tbody>
</table>

Valid test cases 8 characters long

Figure A.2: Results of fuzzing with valid-8 test cases before and after applying countermeasures
Valid test cases 10 characters long

Figure A.3: Results of fuzzing with valid-10 test cases before and after applying countermeasures
Figure A.4: Error Propagation Scenario of segmentation fault
Figure A.5: Error Propagation Scenario 2 of Integer divide by zero exception
Appendix B

B.1 Methodology for Analysis of Results

In this section, we discuss the methodology in which way we analyze the results obtained after running the tests using our security evaluation framework. The following diagram (Figure B.1) would represent the steps that are required accompanied by the files that are generated as a result.

![Methodology for Analysis of results diagram]

Using the assessor i.e., our log reports we check if there was a crash and for what reason the crash occurred. The reason would be identified with the help of debug logs that keep an eye on the input values on every asset.

**Step 1 Look for faults if any that might have crashed the execution of the SUT**

The summary report is generated of the purpose of reducing effort to locate and confirm fault types. The summary report give a total count of every fault type that occurred during the execution of the SUT with test cases. From the report, we can confirm that there is the presence of vulnerabilities in the SUT if there is hangs, segs, timeout or exception. A snapshot of a summary report is shown below.

- total hangs: 0
- total segs: 195
total timeout: 0
total exception: 1
total no failure: 4

To aid the identification of faults, we use the program output in the Assessor. An example of run of a test case by SUT without any crash is given below:

In main()!
CAN_PTO_STATUS: 0100000000000000
CAN_VEHICLE_SPEED:0000000000000000
CAN_THROTTLE_DATA:0000000000000000
CAN_PTO_STATUS: 0000000000000000
CAN_VEHICLE_SPEED:0000000000000CEB
CAN_THROTTLE_DATA:0000000000000000
CAN_PTO_STATUS: 0000000000000000
CAN_VEHICLE_SPEED:0000000000000CEB
CAN_THROTTLE_DATA:0000000000000000

An example of run of a test case by SUT with crash is given below:
In main()!
CAN_PTO_STATUS: 0100000000000000
CAN_VEHICLE_SPEED:0000000000000000
CAN_THROTTLE_DATA:0000000000000000

Step 2 Identify the test case with the fault type that caused the crash
Inside the collector the test cases with their fault type are accumulated in their respective storage space. For example, test_case10 with segmentation fault would be translated from binary to text format and stored inside the “segs” storage folder.

Step 3 Locate the test case at fault
From the debug log file “logs1-200.log” we find those test cases at fault with their corresponding log report. From the crashing point and a short description by the compiler, it is possible to guess the type of vulnerability.
**Example:**

timeout: the monitored command dumped core
FuzzPh1/Invalid/pass0/test1-200.sh: line 11: 2975 Floating point exception

timeout: the monitored command dumped core
FuzzPh1/Invalid/pass0/test1-200.sh: line 11: 2965 Aborted

**Step 4 Identify the line at failure**

To identify the exact line of failure, we jump into the line of code which last executed before the crash, and investigate thereon. We use the default call graph which denotes correct execution. The default call graph should be compared with the call graph that caused failure. Then it is easy to locate the cause of the crash, and then investigating that function manually narrows down the search space.

**Step 5 Identify the unsafe instance**

Investigation with the knowledge that the security tester gained during the creation of the Vulnerability and threat database, should be used if one of the unsafe instances in the Vulnerability and threat database matches with the one identified by the security tester from the results of investigation. If there was no match, then the security tester has to rely on injection of more manual debugging statements in the adjacent areas of the line at failure of the vulnerable function and rerun fuzzing to identify and confirm the unsafe instance.

**B.2 Methodology for Choosing and Applying Countermeasure**

In this section, we discuss the methodology for choosing and applying countermeasure in our security evaluation framework. Finding critical assets in the code i.e. one of the leaf nodes of the AUTOSAR software stack for example a function or a variable is crucial for choosing the countermeasure and determining the exact values allowed, e.g., assets in the representative code are speed, speed Length, throttle angle etc.

Again, we could also gain some hints or the exact solutions from the static analysis tool depending on the kind of the static analysis tools used.
Types of Countermeasures
The countermeasures that we have used are replacement with a safer library function against the existing unsafe library function and range check on assets.

- Exactly replacing the unsafe instance with available safe library instances.
  For example, vulnerable atoi could be replaced by strtol, memcpy with memcpy_s.
- Else exercising additional range check could help to identify overflows. We may read the program specification manual to get those asset values. Then check if that range was overflowed or under flowed. If that turns out true, then we could put a safe exit code or handle it differently. Else allow the next instruction to execute.

Secure Programming Practices
The strategy for choosing and applying countermeasure is dependent on the secure programming practices like checking the minimum and maximum range at the destination buffer, keeping an eye on the inputs i.e., properly sanitizing the inputs before moving to the next instruction etc. Making underflow or overflow checks on every asset usage could be a good choice of avoidance of vulnerabilities at the early stage of source code development.

Unsafe Instance types and Countermeasures
Here we discuss briefly the methodology of choosing and applying countermeasure. Generally, in the code unsafe instances may be a C library function, a user defined function, and a user defined variable or an input argument. The chart below (Figure B.2) describes the countermeasure strategy that can be adopted for each of these types of unsafe instances in the source code of a software.
If one or more of the assets are overflowed again, then we might have missed an important countermeasure for an unknown vulnerability. We could plan to go a step back to previous version of the code with the countermeasure that we last applied, then check from the debug logs where the actual overflow was occurring. After identifying the location of the vulnerability, then apply that important range check condition there and compile and rerun the next fuzzing iteration on the same test inputs as of the previous iteration. If there are value failures we might have to repeat the process again until they are all fixed.

**Error Propagation and Countermeasure adoption**

From the Vulnerability and Threat Database we would take an unsafe instance that was confirmed exploitable and had overflowed in the fuzzing step. Then we would investigate in the code keeping the call graph as an aid to find the source of the vulnerability at the root level. We would traverse with one responsible element as a checkpoint resource that may overflow the buffer. It is worth to mention that we have a confirmed segmentation fault at this stage due to this unsafe instance in the fuzzing step and the next step is to find a suitable countermeasure that can fix the problem permanently at root level.
We would also look into the places in the code that need attention for applying countermeasures. Side by side the debugging areas that could partially cover the vulnerable area adjacent to the unsafe instance.

### B.3 Benefits of Repetitive Fuzzing, an example

In this section, we will show the benefits of repeated fuzzing. By repetitive fuzzing, we found two scenarios with error propagation. In the scenario 2 (see Appendix A, Figure A.5), the example of an exception that we discussed in section 5.2.4, encountered “Integer Divide by zero” fault in the code. We identified the location in the call graph where the fault occurred. Then investigated for error propagation, applied a countermeasure and repeated fuzzing again. After implementing a solution, i.e. applying a countermeasure in the “CalcSpeedFromTachoData()” instance, we found another vulnerability occurring in the result. For example, after we handled the exception, we found that the value of a variable “speed” has suddenly started to fluctuate between largely negative or largely positive number range. This implies, our previous countermeasure was not in an appropriate place and that influenced the speed value, which is not a correct solution by principle.

Therefore, we generated the call graph for this scenario (see Figure A.5). With the help of this call graph, we repeated investigating with more debugging statements at places where we found the inputs are flowing in the code. Lastly, we found that due to improper sanitization of input at the beginning of the main function where the input is received, for example, in asciiCharToUnsignedChar() function, all problems were occurring because of this invalidated inputs. Hence, we systematically prove that with the help of repeated fuzzing and error propagation we can find the root cause of vulnerability that arise from malformed input, which is true in our experiment. We also claim, by this approach, we can find unknown vulnerabilities in code, and can also find systematically appropriate locations for applying countermeasures in the code (for complete illustration see Observation III under section 5.3).
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Rahul Kumar Dutta.