Program Transformations for Vulnerability Detection in Binary Executable Files

Mansour AL-Qattan

PhD

This thesis is submitted in partial fulfilment of the requirements of

De Montfort University for the award of

Doctor of Philosophy

Supervised by

Dr. Feng Chen and Dr. Martin Ward

Software Technology Research Laboratory

Faculty of Technology

De Montfort University

Leicester, UK

April 2017
Declaration of Authorship

I, Mansour Alqattan, declare that this thesis titled, “Program Transformations for Vulnerability Detection in Binary Executable Files” and the work presented in it is my own and has been produced by me as the result of my own original research. I further confirm that:

- This work was done wholly while in candidature for a research degree at De Montfort University during the period of October 2011 to April 2017;
- It has not been submitted, either wholly or substantially, for another degree of this University, or for a degree at any other institution;
- Where I have consulted the published work of others, this is always clearly attributed and cited;
- I have clearly signalled the presence of quoted or paraphrased material and referenced all sources. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help.

Signed: _____________________________________

Date: _______________________________________

i
Publications


Abstract

The major causes of threats in computer systems across the globe have been attributed to vulnerabilities in the underlying software on any hardware platform. Vulnerability analysis entails the process of determining whether a system contains a flaw which an attacker can exploit to compromise the system. Across the years, many approaches and strategies to perform vulnerability analysis and detection have been developed for high-level programming languages but little research has been carried out to detect vulnerability in binary executable codes. This remains a major challenge that must be addressed in the computer science community. This is important given that the first point of attack by hackers is through the binary data level. It is the level through which a potential hacker establishes security loopholes and secret doorways that are not easily detected in low-level programming languages. As such, in an effort to curb the threats from vulnerability, software security developers require better and improved methods and models to analyse binary executable files. Currently, the type of vulnerability tools available to software developers remains fragmented and cannot perform detailed analysis with the desired level of accuracy. In order to aid the detection of vulnerabilities in low-level language by software developers, this research proposes a novel systematic approach for detecting vulnerabilities at the binary level. The approach taken was based on static vulnerability analysis, which entails the re-engineering of low level language by leveraging on the abundant transformation techniques such as FermaT transformation system within Wide Spectrum Language (WSL) program. This research highlights a novel approach and contribution for the extension of existing WSL to enhance its vulnerability detection capabilities using two techniques namely: (i) program slicing combined with static taint analysis and (ii) static single assignment combined with value range analysis. The developed model provides software developers with a new approach of vulnerability detection mechanism in binary executable files. The usefulness of the approach was tested using several scenarios to validate its output and a high level accuracy in the detection pattern was observed. The results show that possible vulnerabilities can be detected in binary executable files.
Dedication

To my parents, my brothers, my sisters and my friends for their love, support and encouragement during this time of challenges.
Acknowledgments

For many years, I have dreamt about receiving a PhD degree and thankfully I have been able to achieve such feat. To this end, my profound gratitude goes to a number of people who assisted me in attaining this goal.

First of all, all thanks and praises be to Allah for granting me sound health throughout my PhD period and always. My profound gratitude goes to my supervisor, Dr. Feng Chen who availed me the opportunity to pursue a doctoral degree. His unfaltering patience, meticulous and inspiring advice provided me the unrelenting courage to strive through one of the most challenging periods of my life to become one of my most important times as well. I am forever grateful for his leading role in fostering my academic, professional, and personal growth.

I express my deep appreciation and gratefulness to my second supervisor Dr. Martin Ward, for his invaluable advice and constant support throughout my study period. I am sincerely grateful to Professor Hussein Zedan for his precious comments and useful suggestions about my work. I appreciate Dr. Helge Janicke for his help with my work on various occasions. I feel so blessed to have been avail the opportunity to have intellectually stimulating discussion with all of you. I sincerely thank the members of staff of the Research Office at the De Montfort University for their outstanding support and management during this dreadful journey towards a doctoral degree.

A great many thanks go to my colleagues in the Software Technology Research Laboratory (STRL): Nayf Alshimiry, Bader Aloufi, Tariq Aljammaz, Hani Alghaiz and Sultan Alhussainy many others, for their outstanding support, valuable suggestions, encouragement and useful discussions. Finally, I am particularly grateful to my parents and my siblings from the bottom of my heart - no words are adequate to express my appreciation for their love, encouragement, patience, prayers and support throughout my life. Their love empowers me to steer through the ups and downs of my doctoral pursuits. This doctoral thesis is dedicated to them.
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>ASM2WSL</td>
<td>Assembly Language to Wide Spectrum Language</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>NVE</td>
<td>National Vulnerability Database</td>
</tr>
<tr>
<td>BC</td>
<td>Boundary Checking</td>
</tr>
<tr>
<td>BOF</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>C2WSL</td>
<td>C Language to Wide Spectrum Language</td>
</tr>
<tr>
<td>C2ASM</td>
<td>C Language to Assembly</td>
</tr>
<tr>
<td>CG</td>
<td>Call Graphs</td>
</tr>
<tr>
<td>COBOL</td>
<td>Common Business-Oriented Language</td>
</tr>
<tr>
<td>CSSV</td>
<td>C String Static Verifier</td>
</tr>
<tr>
<td>DOS</td>
<td>Disk Operating System</td>
</tr>
<tr>
<td>FTS</td>
<td>FermaT Transformation System</td>
</tr>
<tr>
<td>FME</td>
<td>FermaT Maintenance Environment</td>
</tr>
<tr>
<td>FST</td>
<td>FermaT Slicing Transformation</td>
</tr>
<tr>
<td>FTM</td>
<td>Formel Transformation Method</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>FTT</td>
<td>Formel Transformation Technique</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GPL</td>
<td>GNU General Public License</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>ISC</td>
<td>International Information Systems Security Certification Consortium</td>
</tr>
<tr>
<td>ITS4</td>
<td>Implementing Technical Standards</td>
</tr>
<tr>
<td>M</td>
<td>Matrices</td>
</tr>
<tr>
<td>MA</td>
<td>Memory Analysis</td>
</tr>
<tr>
<td>MFFV</td>
<td>Vulnerability of Flaw Function Misusing</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>Open Secure Shell Protocol</td>
</tr>
<tr>
<td>PT</td>
<td>Program Transformation</td>
</tr>
<tr>
<td>RATS</td>
<td>Rough Auditing Tool for Security</td>
</tr>
<tr>
<td>S</td>
<td>Slicing</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
<tr>
<td>TA</td>
<td>Taint Analysis</td>
</tr>
<tr>
<td>VA</td>
<td>Vulnerability Analysis</td>
</tr>
<tr>
<td>VCG</td>
<td>Vulnerability Cause Graph</td>
</tr>
<tr>
<td>VD</td>
<td>Vulnerability Detection</td>
</tr>
<tr>
<td>VDA</td>
<td>Vulnerability Detection Algorithm</td>
</tr>
<tr>
<td>VRA</td>
<td>Value Range Analysis</td>
</tr>
<tr>
<td>WMAT</td>
<td>WSL Memory Analyser Technique</td>
</tr>
<tr>
<td>WSL</td>
<td>Wide Spectrum Language</td>
</tr>
<tr>
<td>WVAT</td>
<td>WSL Vulnerability Analysis Technique</td>
</tr>
</tbody>
</table>
# Table of Contents

Declaration of Authorship.............................................................................................................. i
Publications.......................................................................................................................................... ii
Abstract ................................................................................................................................................ iii
Dedication............................................................................................................................................... iv
Acknowledgments.............................................................................................................................. v
List of Acronyms .................................................................................................................................. vi
Table of Contents ............................................................................................................................... viii
List of Figures........................................................................................................................................ xii
Listing .................................................................................................................................................. xv
List of Tables......................................................................................................................................... xvi

Chapter 1 Introduction ........................................................................................................................ 1
  1.1 Background................................................................................................................................... 2
  1.2 Research Motivation...................................................................................................................... 4
    1.2.1 Address Security Issues in Binary Executable Files................................................................. 4
    1.2.2 Minimise Vulnerabilities in Low level Language................................................................. 4
    1.2.3 Employ Program Transformation into Vulnerability Detection ........................................... 5
  1.3 Aim and Objectives...................................................................................................................... 5
  1.4 Statement of the Research Problem............................................................................................ 6
    1.4.1 Research Questions.................................................................................................................. 7
    1.4.2 Research Scope and Procedural Approach............................................................................. 7
  1.5 Original Contributions................................................................................................................ 8
  1.6 Criteria of Success....................................................................................................................... 9
  1.7 Research Methodology............................................................................................................... 10
  1.8 Thesis Outline............................................................................................................................ 12

Chapter 2 Background and Literature Review.................................................................................... 13
  2.1 Overview of Legacy Systems ........................................................................................................ 14
    2.1.1 Overview of Binary Executable Files .................................................................................. 16
  2.2 Software Vulnerabilities .............................................................................................................. 17
7.6 Discussion of Further Results Obtained from the Detection Tool Developed........130
7.6.1 Test Vulnerability with FermaT Metrics ...................................................136
7.7 Comparison of Output Results from this Work with Other Past Works ............137
7.7.1 Cqual ..................................................................................................137
7.7.2 Goanna ...............................................................................................138
7.7.3 SPLINT ...............................................................................................138
7.7.4 Vulncheck ...........................................................................................138
7.7.5 VSA-Based Approach ..........................................................................139
7.7.6 BOON .................................................................................................139
7.7.7 PolySpace ............................................................................................140
7.7.8 VulMiner ..............................................................................................140
7.7.9 BitBlaze ...............................................................................................141
7.7.10 Detecting Known and Unknown Vulnerabilities .....................................142
7.7.11 Discussion ...........................................................................................143
7.7.12 Dataset Selection ................................................................................144
7.7.13 Vulnerability Detection .......................................................................144
7.8 Summary ..................................................................................................145
Chapter 8 Conclusion and Future Work ............................................................146
8.1 Thesis Summary and Concluding Remarks ................................................147
8.2 Major Contributions of the Current Research .............................................149
8.3 Success Criteria Revisited ..........................................................................150
8.4 Research Limitations ................................................................................152
8.5 Future Work ..............................................................................................153
List of References ............................................................................................155
Appendix A Comparison Vulnerability Detection Tools ....................................165
Appendix B Define Other Components Related to System Architecture .............168
Appendix C Additional Information on Process of Vulnerability Detection ..........174
Appendix D Prototype of Vulnerability Detection .............................................187
Appendix E Strategies Associated with Vulnerability Detection Model .............209
List of Figures

Figure 1-1: Trends in Software Vulnerabilities (reproduced from Zhang et al. 2016) ................. 2
Figure 2-1: Vulnerability Cause Graph (VCG), Adapted from Byers et al., (2006) ....................... 22
Figure 2-2: Security Goal Indicator Tree (Adapted from Peine et al. (2008)) ...................... 32
Figure 2-3: Security Activity Graph, SAG, Adapted from Byers and Shahmehri (2008) ........... 33
Figure 2-4: Reverse Engineering Procedure ................................................................. 36
Figure 2-5: Overall Pictorial Representation of Software Re-engineering .............. 37
Figure 3-1: Flow Chart for Vulnerability Detection within the Overall Model ....................... 48
Figure 3-2 Steps Involved in Vulnerability Detection .................................................. 50
Figure 3-3: Research Design and Structure ................................................................. 51
Figure 3-4: Overall Flow Chart Illustrating the Key Steps Involved in Vulnerability Detection Based on Contributions from the Current Work .................................................. 52
Figure 3-5: Pictorial Representation Translation for Vulnerability Detection Based ....... 54
Figure 3-6: Overall System Framework for Vulnerability Detection .............................. 55
Figure 3-7: An Illustrative Example how Source Code Transform between Different Programming Languages ......................................................................................... 56
Figure 4-1: Architecture Illustrating the Vulnerability Detection Process ......................... 60
Figure 4-2: Process of Translation .................................................................................. 62
Figure 4-3: Abstract Syntax (AST) Tree Representation of C = A + B ......................... 63
Figure 4-4: Taint Analysis Process ................................................................................ 64
Figure 4-5: The Transformation Process for Restructuring ........................................... 65
Figure 4-6: Process for Vulnerability Detection .......................................................... 66
Figure 4-7: Components of Vulnerability Analyser ...................................................... 67
Figure 4-8: Framework for WSL Memory Analyser Technique .................................... 70
Figure 4-9: Illustration of the Mode of Operation of the Proposed Method for Memory Analyser ................................................................................................................. 74
Figure 4-10: Calculation Procedure of the Effective Size of a Stack Memory ................. 74
Figure 4-11: Framework for Decision Maker Phase ..................................................... 75
Figure 5-1: Memory Frame for Vulnerability Assembly Code ....................................... 86
Figure 5-2 Output of the Vulnerable Code in Assembly ............................................... 87
Figure 5-3: Memory Frame for Non-Vulnerability Assembly Code ............................... 88
Figure 5-4: The Output after the WSL Transformation .................................................. 91
Figure 5-5: Shows the Slicing Method of the Detection in WSL ...................................... 93
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-1</td>
<td>Program Transformation</td>
<td>179</td>
</tr>
<tr>
<td>D-2</td>
<td>Development Tool Prototype Design of Vulnerability Detection</td>
<td>187</td>
</tr>
<tr>
<td>D-3</td>
<td>Simple Illustration of Vulnerability Detection Framework Adopted in this Work.</td>
<td>190</td>
</tr>
<tr>
<td>D-4</td>
<td>Display of Parser Log File.</td>
<td>191</td>
</tr>
<tr>
<td>D-5</td>
<td>Illustration of a Successful Translation</td>
<td>191</td>
</tr>
<tr>
<td>D-6</td>
<td>Running the code an input “this string is too long”</td>
<td>192</td>
</tr>
</tbody>
</table>
Figure D- 7 The Output of Hacked Code after the Translation.................................192
Figure D- 8 Call Graph of WSL Code.................................................................193
Figure D- 9 Call Graph and Control-flow Graph in Assembly..............................194
Figure D- 10 The Output of WSL Code after the Translation.................................195
Figure D- 11 The Process of Transformation as in (a), (b), (c) and (d)....................196
Figure D- 12 The Call Graph After the Transformation........................................197
Figure D- 13 Abstract Syntax Tree in WSL..........................................................199
Figure D- 14 Call Graph Showing how Syntactic Slicing was Applied within the Codes.........200
Figure D- 15 Call Graph after Slicing....................................................................201
Figure D- 16 Abstract Syntax Tree for the Vulnerable Code.................................202
Figure D- 17 Call Graph after the SSA Transformation Process...........................202
Listing

Listing 2-1 Vulnerability Code in C Language ........................................................................................................ 19
Listing 2-2 Fragment of Vulnerability Code ........................................................................................................... 21
Listing 4-1 GCC Decompile Commands ................................................................................................................. 61
Listing 4-2 Model the assembly instructions and memory to WSL. ................................................................. 65
Listing 4-3 Model the assembly instructions and memory to WSL. ................................................................. 68
Listing 4-4 Fragment of WSL Code ......................................................................................................................... 71
Listing 4-5 Applied the SSA Transformation to the Code .................................................................................... 71
Listing 4-6 Model the assembly instructions and memory to WSL. ................................................................. 72
Listing 4-7 Formalisation of Memory Analysis ..................................................................................................... 72
Listing 4-8 Heap Command .................................................................................................................................. 72
Listing 4-9 Model the assembly instructions and memory to WSL. ................................................................. 75
Listing 5-1 More Secure Code of Assembly ........................................................................................................... 84
Listing 5-2 Size Of Variable Determined in the Location ..................................................................................... 85
Listing 5-3 Disassembly of Flawed gets() and strcmp() Calls in Test Program ............................................ 85
Listing 5-4 Created Memory Frame which Caused Vulnerability in Assembly ................................................. 86
Listing 5-5 Disassembly of more Safe fgets() and strncmp() Calls in Test Program .................................... 87
Listing 5-6 Output of Secure Code in Assembly .................................................................................................. 88
Listing 5-7 WSL Action system ............................................................................................................................. 89
Listing 5-8 Commands to Run the FermaT Tool .................................................................................................. 90
Listing 5-9 WSL Code .......................................................................................................................................... 91
Listing 5-10 Renamed Array References as Separate Variables ....................................................................... 96
Listing 5-11 Detected the Overwritten Variables by Slicing with Pointed Taint Position ......................... 96
Listing 5-12 Detected the Relevant Code by Using Slicing Transformation with Taint Analysis 97
Listing 6-1 Command to Run the Translation Tool ............................................................................................... 107
Listing 6-2 Command to Run the Transformation Engine .................................................................................. 108
Listing C- 1 Adaptive Assembly Translator Specifications ................................................................................. 176
Listing C- 2 Adaptive Process of Translator ....................................................................................................... 177
Listing C- 3 Meta-WSL Code for Transformation ............................................................................................... 182
Listing D- 1 Example of Slicing ........................................................................................................................... 199
List of Tables

Table 4-1 VCs Useful for Detecting Vulnerable Code. .......................................................... 68
Table 4-2 Vulnerability Analyser x Memory Analyser............................................................ 76
Table 5-1 Vulnerability Features............................................................................................. 80
Table 5-2 Assembly Vulnerability Rules.................................................................................. 80
Table 5-3 Illustrate Example with and without Vulnerability. .................................................. 81
Table 5-4 Illustrate Disassembly Code of the C Program. ....................................................... 83
Table 5-5 Illustrate the Differences between Vulnerability Pattern and non-Vulnerability Pattern ......................................................................................................................... 84
Table 5-6 Detected Vulnerability Pattern in the Tested WSL Program ................................... 94
Table 5-7: The Position Point of each Assignment Statement ................................................ 95
Table 5-8: Detecting Vulnerable Code with Syntactic Slicing Method..................................... 97
Table 5-9 Detecting Method with Semantic Slicing ................................................................. 98
Table 5-10 VA Output an SSA Process for Detecting Vulnerability ....................................... 100
Table 6-1 Example of Translation Command from ASM to WSL ......................................... 107
Table 7-1: Weight for Each Pattern......................................................................................... 117
Table 7-2: VCs Weight for Detecting Vulnerable Code. ......................................................... 118
Table 7-3 Weight of the Total Detections. ............................................................................. 118
Table 7-4 Vulnerability Pattern in C with Equivalent Pattern in WSL .................................. 120
Table 7-5: Test Result for Normal Processes. ........................................................................ 121
Table 7-6 Vulnerability Code in Assembly............................................................................. 122
Table 7-7: Test Vulnerabilities for Theory Validation.............................................................. 123
Table 7-8: Evaluate Effectiveness of Vulnerability Detection with the ASM2WSL Output .... 124
Table 7-9: Evaluate Effectiveness of Vulnerability Detection with the C2WSL Output .......... 125
Table 7-10: Vulnerability Pattern Detection for 20 Examples................................................. 126
Table 7-11: Verification of the Result for all Tools Developed................................................. 127
Table 7-12: Results of Running Vulnerability Analyser Based on Taint Analysis and Slicing.... 131
Table 7-13: Results of Running WVDT in Value Range with SSA......................................... 132
Table 7-14: Results of Running WVDT Involve Taint Analysis with Slicing and Value Range with SSA Transformations. .................................................................................. 132
Table 7-15: Evaluation Result on Effectiveness........................................................................ 133
Table 7-16: Evaluation Result on Efficiency............................................................................. 134
Table 7-17: Level of Accuracy of Each Data Set. ..................................................................... 135
Table 7-18: Overall Correct Detection Rate (OCDR) with Integrity Method. ........................................135
Table 7-19: Matrices before the Slicing. ..................................................................................................136
Table 7-20: Matrices after the Slicing. ..................................................................................................137
Table 7-21: Results of Running Cqual Program Detected False Positives False Negatives ..........137
Table 7-22: Results of Running Goanna ..............................................................................................138
Table 7-23: Results of Running SPLINT. .............................................................................................138
Table 7-24: Results of Running Vulcheck with Taint Analysis and Value Range Propagation Mode. .................................................................139
Table 7-25: Results of Running VSA-Based Approach .................................................................139
Table 7-26: Results of Running BOON ...........................................................................................140
Table 7-27: Results of Running PolySpace. .......................................................................................140
Table 7-28: Results of Running VulMiner ........................................................................................140
Table 7-29: Results of Running BitBlaze ........................................................................................141
Table 7-30: Summary of Comparison between the Current Work (WVDT) and Other Works 141
Table A- 1 Summary of the Analysis Based on Vulnerability Detection Work of Researchers. 165
Table A- 2 Classification of the Analysis Based on Vulnerability Detection ........................................166
Table A- 3 Performance Comparison of the Analysis Based on Vulnerability Detection ..............167
Table B- 1 WSL Code Program ........................................................................................................169
Table B- 2 Result of the Compiler in Scheme Language .................................................................170
Table C- 1 Purpose of Transformation Rules ..................................................................................180
Table C- 2 Potentially Vulnerable Functions and Language Constructs ........................................183
Table D- 1 Vulnerability Patterns ..................................................................................................187
Table D- 2 Classification of Vulnerability Data ...............................................................................188
Table D- 3 the Common Weakness Enumeration (CWE) ..............................................................189
Table D- 4 The Implementation Process Abbreviation .................................................................190
Table D- 5 The Output of the Translation Process. .......................................................................191
Table D- 6 Process of Transformation ............................................................................................197
Table D- 7 Description of Key Metrics for Translations within WSL ..............................................198
Table D- 8 Description of Key Metrics for Translations within C2WSL and ASM2WSL ..........198
Table D- 9 Vulnerability Slicing to Detect the Vulnerabilities ........................................................201
Table E- 1 Base Strategies Associated with Vulnerability Detection ...........................................209
Chapter 1 Introduction

Objectives

- To provide the motivation for this research.
- To formulate research questions.
- To define the scope of thesis.
- To highlight the original contributions to knowledge.
- To summarise the research methodology.
- To outline the thesis structure.
Chapter 1: Introduction

1.1 Background

Globally, vulnerability in software is generally regarded as a security flaw or weakness, or a programming glitch found in software or an operating system which can degenerate into wider security implications when exploited by an attacker with a malicious intent to alter the normal pattern of the system (Jimenez et al., 2011). Today, there is hardly any high level business that does not make use of application software with the view to run more efficiently, given that software powers everything ranging from critical infrastructure and transportation to healthcare, commerce and financial systems (Veracode, 2013). As such, as the number of software increases in the market, the number of vulnerabilities also increases (Jimenez et al., 2011). This growing dependence on software has the competitive edge of improving efficiency of business operations but sometimes it can also come at a cost.

![Figure 1-1: Trends in Software Vulnerabilities (reproduced from Zhang et al. 2016)](image)

On a yearly basis, a large number of software vulnerabilities are discovered and reported in various applications (Figure 1-1). In fact, in a recent research carried out by U.S. Department of Homeland Security (DHS), it was established that 90% of security breaches and incidents emanated from exploits against vulnerabilities in software (Veracode, 2013). A study published by the International Information Systems Security Certification Consortium (ISC, 2016) through
its Global Information Security Workforce suggests that software vulnerabilities continue to be a major source of concern for security experts but has not been given the requisite priority. This lack of concerted effort towards unravelling software vulnerabilities is attributed to a lack of understanding of the origin of such vulnerabilities. Gaining an understanding of such vulnerabilities can ensure that software vendors are better prepared to establish effective strategies for discovering and correcting vulnerabilities, thereby reducing the associated risks caused by the proliferation of the software.

Vulnerability analysis is an integral component of software development lifecycle (Rawat and Mounier, 2012). It is a process that helps to ascertain whether a software system has vulnerabilities which an attacker can leverage with the view to compromise the underlying software or the platform upon which the system is run (Cova et al., 2006). The overall focus of vulnerability detection and analysis as with other forms of security breach detection mechanism is on the identification and correction of flaws. A practical example of such vulnerabilities is buffer overflow or buffer overruns which is regarded as the third most dangerous software error (Martin et al., 2011). Buffer overflow is a form of vulnerability whereby a program while writing data into a buffer, overruns the boundary of the buffer by overwriting the adjacent memory locations (Christopher, 2002). Such overflow can be employed to introduce malicious codes, altering the sequence of execution of the targeted software whilst executing the injected malicious code and taking control of the software.

Over the years, a number of strategies for performing vulnerability detection and analysis have been developed to target high-level programming with little research focusing on low-level language such as assembly language. In part, this is because many computer systems (e.g. legacy systems) which run on assembly languages were developed at a time when there was less concern about security issues and vulnerabilities. Moreover, identifying vulnerabilities in such systems can be expensive, given the dearth of assembly language programming experts and the prohibitive costs of hiring the few available ones. Given that the first point of attack by hackers is through the binary data level where security loopholes and secret doorways to the underlying platforms of software are exploited, it is important to address this problem through the development of vulnerability detection tools focusing on low level language.

The uses of program transformation techniques are useful as potential strategy for vulnerability detection. Wide Spectrum Language (WSL) is an example of such program transformation technique that can be used to aid the detection of vulnerabilities by acting as an intermediary between low-level and high-level programming languages. However, WSL is not designed for directly detecting vulnerabilities in assembly language, thereby requiring some level
of extension and refinements. Against this backdrop, the current research work seeks to develop a novel systematic technique for vulnerability detection at the binary level of assembly language programming. This will be achieved through the re-engineering of assembly language programming by leveraging the FermaT transformation mechanism within WSL using program slicing technique combined with taint analysis on one hand and static single assessment combined with value range analysis on the other hand. The integrated approach taken offers simplicity and additional refinement through the reduction of the number of lines of code thereby increases the speed of vulnerability detection. In doing so, it is intended that the model developed will provide software developers with an efficient and effective approach for detecting vulnerabilities in low level language programming with a focus on binary executable files.

1.2 Research Motivation

The motivation for embarking on this work stems from a number of interrelated observations that include the following:

1.2.1 Address Security Issues in Binary Executable Files

As the modern ICT-based society witnessed a tremendous growth and expansion through the advent of computer networks, distributed systems and the Internet, ensuring security of critical infrastructure which depends on the aforementioned system expansion has led to important regime of research focusing of computer security (Kwon et al., 2017). This is especially important given that most systems are vulnerable and such vulnerabilities are the first point of attack and exploitation. As such, vulnerability detection has become a major concern in the computer science and software engineering parlance. Given that most vulnerability is exploited through binary executable files of low-level programming languages; attention is focused on them given the non-availability of the source code of the programs for which vulnerability is to be detected. It is therefore important to develop practical approaches and models/tools supported with robust theoretical underpinnings to aid the detection of vulnerabilities in binary executable files.

1.2.2 Minimise Vulnerabilities in Low level Language

Generally, incorporating security features such as vulnerability detection capabilities into traditional legacy systems powered by low level programming language is an arduous task due to a number of factors including limited understanding of the associated risk and complexities, increased defects as well as high development costs/cycles. All these factors constitute major
Chapter 1: Introduction

sources of latent security flaws and their chances for continuous occurrence have increased tremendously. To overcome these challenges and manage the associated risks in a cost-effective manner, highly automated security assurance assessments will need to be performed. To this end, the current work seeks to develop such automated vulnerability detection scheme to address these challenges.

1.2.3 Employ Program Transformation into Vulnerability Detection

The need to re-engineer existing software is to ensure that its source code is well understood, with better control strategies in place to detect vulnerabilities. Such objectives are fulfilled using program transformation as a core technique. Within the research community and industrial settings, improvements focusing on the efficiency and automation of program transformation have become a major area of concern. With the increasing demand and desire to retrofit legacy systems through their binary executable files, their evolution is becoming a matter of urgent need. Although some approaches have been introduced for vulnerability detection and analysis in binary executable files, such approaches are not sufficient for understanding underlying codes at the binary level (Kwon et al., 2017). Current tools for program transformation do not provide full solution to the problems associated with vulnerabilities detection in binary executable files. For instance, FermaT which is a transformation technique within WSL and possesses the ability to automatically or semi-automatically restructure the underlying codes of binary executable files. This is because its core, a transformation engine, is based on mathematically proven program transformations and ensures that transformed programs are semantically equivalent to their original state. Despite this program transformation capacity, it lacks the inherent ability to perform vulnerability detection and analysis. There is therefore the need to develop an appropriate mechanism, based on program transformation techniques, with the view to detect potential vulnerabilities in binary executable files especially as it pertains to buffer overflow. This will enhance the efficiency and accuracy of vulnerability detection in low-level programming language such as assembly language programming.

1.3 Aim and Objectives

The central aim of this research is to investigate and utilise program transformation techniques for detecting and analysing vulnerabilities in low level languages such as assembly language programming especially as it pertains to binary executable files using reverse engineering principles and techniques to obtain the source code. Linked to this central aim are the following specific objectives:
Chapter 1: Introduction

(a) Conduct a detailed review of the limitations of existing software vulnerability detection mechanisms, and highlight how such limitations can be improved in the context of low level programming languages.

(b) Develop a new approach for the prediction and detection of vulnerabilities in binary executable files (reverse engineer to low level programming language such as assembly language programming) by using enhanced Wide Spectrum Language (WSL) and its transformation engine, with a specific focus on buffer overflow.

(c) Extend and optimise the existing WSL workbench to enhance its vulnerability detection capabilities with the aid of software re-engineering concepts such as program slicing technique combined with taint analysis as well as static single assignment combined with value range analysis. This entails the identification of vulnerability rules and their enforcement into the overall detection system.

(d) Evaluate the newly developed vulnerability detection model using various scenarios to validate its output in terms of level of accuracy and reliability based on predefined criteria.

1.4 Statement of the Research Problem

Globally, a number of application systems were developed when little attention was paid to security and vulnerability challenges (Campara and Mansourov, 2008). Given the changes in business dynamics and computer security requirements over time, the need to take security seriously has become important. Vulnerability detection and analysis techniques have been developed for high-level programming languages with a lot of success recorded but little research has been conducted regarding such detection in low level programming language. This is largely due to the fact that most systems powered by low-level programming language lack proper documentation and source codes but possess only binary executable files thereby making vulnerability detection extremely difficult. Additionally, there are no adequate tools to conduct reverse engineering for the purpose of vulnerability detection. Furthermore, there is a dearth of programming experts in low-level programming language, creating further challenges in addressing vulnerability issues. To this end, the current work seeks to develop a novel, approach and optimised vulnerability detection mechanism to address the aforementioned issues.
Chapter 1: Introduction

1.4.1 Research Questions

The development and formulation of a research question can assist in scoping the research to determine the important features of the theme under investigation. In the context of the current research, based on the gaps identified in the review of literature (detailed in chapter two) and the establishment of a problem statement, the questions that the research seeks to provide answers come into focus. The main research question is therefore:

How vulnerability can be detected and analysed in binary executable files powered by assembly language programming, and how systematic detection can be improved to enhance the features of accuracy and reliability?

Based on the main research question stated above, it seems clear that a number of interrelated research activities will be involved and this therefore led to the following sub research questions:

(a) Given the difficulty involved in detecting vulnerabilities in binary executable files powered by assembly language programming, how can the efficiency and reliability of detection mechanism be enhanced?

(b) What are the limitations of the existing methods/models for detecting vulnerabilities in binary executable files of low-level language such as assembly language programming and how can such limitations be overcome?

(c) Literature suggests that Wide Spectrum Language (WSL) program is a powerful tool endowed with many transformation techniques but has limitations in detecting vulnerabilities in assembly programming language when used as an intermediate-level language between high-level and low-level programming languages. Against this backdrop, how can WSL be extended to enhance its vulnerability detection capabilities in binary executable files?

1.4.2 Research Scope and Procedural Approach

The scope of this research is limited to the development of efficient approach for vulnerabilities detection in binary executable file powered by assembly language programming using WSL and its Transformation engine (FermaT), with a specific focus on buffer overflow. It is a research endeavour that employed three programming languages namely C language (a high level language), WSL language (an intermediate level language or formal language) and assembly
Chapter 1: Introduction

language (a low level language) to gain a deep understanding of how vulnerabilities can be detected and analysed in binary executable files. The overall research scope is based on three key steps which provide the procedural framework upon which the research task is accomplished. The first step entails the development of a compiler that translates from C language to WSL (C2WSL) in order to acknowledge the pattern of the vulnerabilities which are easily recognisable and understandable in a high level language such as C. This step basically entails vulnerability analysis and translation. The second step involves development of an algorithm to detect and analyse vulnerability such as buffer overflow in WSL. Such WSL vulnerabilities analysis involves vulnerabilities checking, decision making and transformation (i.e. memory checking (stack and heap), isolation and conversion (i.e. action system to WSL)). The penultimate step entails the development of a compiler to translate from assembly language to WSL (ASM2WSL) in order to acknowledge the pattern of the vulnerabilities. Overall, the compiler adopted in this research was constructed based on the results from C2WSL to ascertain correct translation before comparison is drawn between the outputs C2WSL and ASM2WSL to observe whether the vulnerability detection requirements are met. Full details of the involved steps are provided in Chapter three.

1.5 Original Contributions

- A program transformation based vulnerability detection approach for binary executable files.

This thesis addresses and explains how program transformation methods and techniques can be enhanced and adapted for the purpose of vulnerability detection in the binary executable files of systems powered by low-level programming language such as assembly language programming. This entails a rigorous demonstration of the understanding of software re-engineering principles using practical examples to demonstrate the benefits such principles can offer for vulnerability detection.

- Enhancement of vulnerability detection techniques for binary executable files.

The developed techniques are now termed as WSL Translation for Modelling Vulnerabilities (WTMV) and WSL Vulnerability Detection (WVD), with improved and enhanced both program translation and vulnerability detection mechanism based on transformation techniques to detect and analyse vulnerabilities in binary executable files. Two techniques, namely: (i) integration of program slicing within a taint analysis scheme and (ii) integration of static single assignment with value range analysis, were developed to provide software developers with a robust and reliable vulnerability detection mechanism for binary executable files.
Chapter 1: Introduction

- **Extension of the WSL workbench for vulnerability detection functions.**

  Previous software re-engineering work has focused on restructuring and refinements of the underlying code, however such work is limited in scope given their lack of support for vulnerability detection in the maintainer's tool-set, generating problems during program transformation. In this thesis, a Vulnerabilities detection Mechanism was developed to address the aforementioned shortcomings and problems. This was achieved by extending the workbench functionalities of the existing WSL into an improved tool by leveraging the FermaT transformation engine to aid effective and reliable vulnerability detection. Specifically, this was achieved through the incorporation of boundary checking mechanism and implementation strategy tested by vulnerability detection in binary executable. Further details and summary of the original contribution to knowledge is provided in Chapter 8.

1.6 Criteria of Success

Given the aim of the research which is to investigate and develop adequate techniques for detecting and analysing vulnerability patterns in binary execution file powered by assembly language, the overall success criteria will depend largely on the extent to which this overall aim is realised. This overall aim has been achieved as evident in the subsequent chapters of this thesis. A brief summary of the criteria for success is provided below:

(a) The ability of the developed model in this thesis to efficiently and reliably detect vulnerabilities in binary executable files in assembly language programming through an extended and enhanced WSL framework is a key success criterion.

(b) The achievement of the design of a vulnerability detection mechanism for binary executable files by combining software re-engineering techniques such as program slicing, with taint analysis as well as static single assignment with value range analysis, is an indication that the aim of the thesis has been delivered successfully.

(c) A demonstration that the proposed detection algorithm developed possesses the ability to detect any inconsistency related to the functions, variable data, and pattern matching process, using three phases of detection checks namely (i) vulnerability analyser, (ii) memory analyser, and (iii) decision maker, is a criterion for success.

(d) The achievement of no false positives when prototypes were tested and the fact that the characteristic of vulnerability detection is unique to program vulnerabilities and can be used to distinguish between viral and normal processes is an indication of success delivered in this thesis.
Chapter 1: Introduction

1.7 Research Methodology

At an operational level, a methodology is the structure which details the step-by-step processes through which the current work is carried and it involves the use of specific methods to collect the adequate and representative evidence of a problem (Knight and Cross, 2012), developing appropriate approaches to address the problem whilst demonstrating the validity of any findings (Amaratunga et al., 2002). Generally speaking, there are a number of research approaches/methods including qualitative and quantitative research, action research, experiment-based research, case study research (e.g. exploratory or confirmatory research), empirical research, conceptual research, survey research and many more. Easterbrook et al. (2008) provided a rigorous analysis on the selection of methods for research in software engineering and computer science.

Given that in the computer science or software engineering parlance research projects focus mainly on developing novel methods and systems, the aforementioned research method types may not be adequately applied. In the context of the current work, a research method termed constructive research as described by Dodig-Crnkovic (2010) is adopted. Constructive research method is mainly adopted towards the formulation of solution to a given identified problem through the development of new models, algorithms, frameworks, prototypes, techniques or software. Such research methods can form the basis to original contribution to knowledge in the field of computer science or software engineering. As such, through the development of a new technique to address a given problem in the academic field, a contribution to knowledge, academic and practice can be adjudged satisfactory.

A number of research works have been carried out into the detection of vulnerabilities in binary executable files especially as it pertains to buffer overflow. Therefore, the current research focuses on this issue with two main aims namely: (i) investigation of the loopholes into binary executable files and the identification of the system requirements for vulnerability detection by developing a strategy-based model informed by program transformation techniques; (ii) enforcement of the vulnerability rules into the developed vulnerability model using the requirement model (e.g. program slicing combined taint analysis) and formal program transformation techniques.

The development of vulnerability detection tool for binary executable files in assembly language programming is not an easy task. As such, achieving such feat will require a carefully mapped out methodological framework. A good research methodology offers the developers ample opportunities and alternatives by bypassing the risks of selecting the bad options while
Chapter 1: Introduction

selecting an approach. In the context of the current work, the choice of an appropriate method allows the programmer to focus on vulnerability related issues and complexities in design process so as to avoid the areas where the research can be delayed. The methods adopted through the prism of constructive research method as highlighted above is based on key steps namely:

(a) **Identification of gap in knowledge:** This entails a review of relevant literature, describing the relevant background issues for this research. Accordingly, the review of literature in the specific area of study (vulnerability detection in binary executable files in assembly language programming) was then used to formulate the conceptual framework which guides the current research. The review establishes the link between what has already been researched and what the current study investigates, identifying current gaps in knowledge that this study seeks to fill.

(b) **Development of theoretical and conceptual framework:** The literature review assisted in identifying the key attributes of the problem under investigation which informed the development of the theoretical framework. The theoretical framework was developed through the identification of relevant methods and techniques for vulnerability detection in binary executable files. These identified methods and techniques, which are captured within the theoretical framework, are then adopted to formulate a conceptual framework which drives the current research.

(c) **Identification and categorisation of program transformation techniques:** This entails gaining an understanding of transformation techniques that will be suitable for extending the vulnerability detection capabilities of WSL which is used as an intermediary programming language between low-level language and high-level language. Such an understanding will assist in establishing new transformation rules for WSL based on mathematical analysis. This therefore lays the foundation for the research work.

(d) **Overall development of the vulnerability detection model:** This streamlines the research to the key area of vulnerability detection in binary executable files and how WSL can be enhanced to achieve it. This entails the use of program slicing combined with static taint analysis; and static single assignment combined with value range analysis.

(e) **Model implementation, evaluation, validation and verification:** Here, the overall vulnerability detection model developed is tested across a number of scenarios to ascertain its accuracy and reliability. This is then followed by validation and verification. Verification entails checking whether the vulnerability detection model developed is indeed a true representation of the actual purpose it was designed for (i.e. detecting vulnerabilities...
Chapter 1: Introduction

in binary executable). Validation refers to checking the vulnerability detection model to confirm that it actually represents the main concept that is being modelled. Verification ensures that the system or model is constructed in a right manner and validation entails constructing the right system (O'Keefe et al., 1986).

1.8 Thesis Outline

The remainder of the thesis is organised into seven chapters as follows. In Chapter 2, a review of the existing literature detailing an overview of software vulnerabilities issues is presented. The review presented in Chapter 2 provides the ‘window’ through which the current study is viewed. The objective of Chapter 3 is to present and demonstrate the methodical framework adopted in achieving the research objectives, detailing the steps involved in vulnerability detection using appropriate techniques and program transformation techniques. Chapter 4 provides a detailed description of the proposed novel Vulnerability Detection Algorithm for Low Level Language systems, in terms of its underlying principles and components, including system structure, system requirements and system outputs. Chapter 5 illustrates the outputs based on the implementation of vulnerabilities detection scheme developed. In Chapter 6, the prototype support tools are presented. In Chapter 7, the validation and evaluation results that complement the design and implementation of the vulnerability detection scheme are presented. A summary of the conclusions from the analysis carried out while this research, original contribution to knowledge, limitations of the work and the direction of possible future extension of the research are presented in Chapter 8.
Chapter 2 Background and Literature Review

Objectives

- Provide an overview of structures of legacy systems, their importance and applications.
- Brief explanation of software vulnerabilities and their types.
- Review the literature concerning the software vulnerabilities detection.
- Provide an overview of software re-engineering and software maintenance.
- Provide an overview of program transformation and wide spectrum language.
Chapter 2: Background and Literature Review

This chapter presents a review of extant literature, describing the relevant background issues and literatures for this research. Accordingly, the review of literature in the specific area of study (vulnerability detection in low-level programming language with a focus on binary executable files) was then used to formulate the conceptual framework which guides the current research. The chapter therefore establishes the link between what has already been researched and what the current study investigates, identifying current gaps in knowledge that this study seeks to fill. The chapter commences with a brief overview of legacy systems which are systems that are mostly powered by low-level language such as assembly language programming as presented in the section that follows.

2.1 Overview of Legacy Systems

In chapter one, a brief overview of the main causes of security issues in computer systems especially as it pertains to systems powered by low level programming languages such as assembly language was presented. One of the main areas where assembly language still find applications is in legacy systems. Across the globe, companies such as multinational corporations invest a considerable amount of money on software systems and infrastructure with the view to derive return from such investments. To derive such benefits, companies always anticipate that such software platforms must be usable for a long number of years. However, the lifespan of software systems varies, with some of them remaining in use for more than ten years after its initial development, but some large corporations still rely heavily on software infrastructure that have been in existence for more than 20 years (RALYTÉ et al., 2017). This is because many of such old systems are business-critical and any form of failure of these systems can wreck serious havoc on the day-to-day operations of a business (LÉONARD, 2017). To this end, these old systems have been collectively referred to as legacy systems in software engineering parlance. They are generally regarded as computer systems that are outdated due to the evolution of high level programming language but are still of important use for certain business-critical applications.

Given that the business dynamics and needs of any corporation changes over time due to factors both internally and externally including: changes in market structures, current state of the national and international economies, changes in law and legislations, changes in management structure and structural reorganisation of a corporation, the need to develop new or modified software requirements of legacy systems arises from time to time (LÉONARD, 2017). As such, for businesses to cope with the aforementioned changes, they replace their equipment and software infrastructure with modern systems. However, a complete overhaul of legacy systems
with replacement with new systems can constitute significant business risk for a number of reasons as detailed in LÉONARD (2017).

Accordingly, maintaining legacy systems prevents the risks associated with replacement efforts, but at some point all systems must change to remain useful. As systems get older, they tend to become more expensive during adjustments of their underlying software platform. This is particularly the case for legacy software systems for many reasons such as (LÉONARD, 2017; RALYTÉ et al., 2017): (i) the possibility of components implemented with different programming techniques across the entire system; (ii) some parts or all of the system may be written using programming languages that are considered obsolete and finding people with expertise in such obsolete programming languages can be very difficult. This may lead to an expensive outsourcing of maintenance of the legacy system; (iii) documentation of systems specification is sometimes insufficient and obsolete. In some instances, the only available documentation is the original system source code in its executable version; (iv) corruption of the overall structure of the legacy systems resulting from several years of maintenance activities; (v) optimisation techniques adopted to save space or increase execution speed could hinder comprehensibility of the system, causing difficulties for programmers who are only conversant with modern software engineering protocols and who do not possess knowledge of the prior programming techniques adopted; (vi) structures of files used during maintenance and upgrade may be incompatible with each other, due to duplication of data or inaccurate or incomplete or in some instance out of date dataset.

Given the presentations above, it readily becomes obvious that businesses that utilise large number of legacy systems as part of their functional business activities will be plunged into a fundamental conundrum. For example, if they stick to their current legacy infrastructure and embark on changes when it is required, their overall cost will increase eventually. At the same time, if they elect to substitute their legacy systems with modern systems, it will be expensive and even at that, such modern systems may not guarantee an effective and efficient business support as derived from the existing legacy systems. Against this backdrop, corporations whose bulk of their systems are still built on legacy platform are investing in software engineering concepts and techniques that are endowed with the possibility of extending the lifespan of legacy systems without compromising performance whilst guaranteeing reduced costs in maintaining the system in use. Summarily, the replacement or maintenance of a legacy system is both risky and expensive. The need to address these issues through the development of cost-effective strategies is one of the hallmarks of the current research work. In the section that follows, a brief review of
avenue through which a legacy system can be exposed to attack due to vulnerability stemming from binary executable files is presented.

2.1.1 Overview of Binary Executable Files

In chapter one, it was highlighted that one of the key objectives of the current work is to develop techniques or models for vulnerability detection and analysis in binary executable files. The focus is on binary executable files given that they are the first point of attack by hackers seeking to exploit the vulnerabilities of a system. A binary file is generally a file that contains information that must be interpreted by a program or a microprocessor with the capabilities to understand in advance how the information is structured and formatted. Generally speaking, executable (i.e. ready-to-run) programs are often known as binary files with “.bin” used to represent the file name extension. More specifically, a binary executable represents the machine code equivalent of any programming language (e.g. high-level or low-level language) which has been assembled or compiled to become integral to the overall functioning of a particular operating system or software platform (RAMAN et al., 2017). Within different operating systems, the overall format of binary executable files varies widely. For instance, Windows NT, Windows 98 and Windows 95 operating systems make use of Portable Executable (PE) format which entails detailed information regarding the different aspects of the program which form the integral part of the file. The PE format contains a number of segment including: (a) .text which encompasses the code and the point of entry of the application; (b) .data, which encompasses different forms of data; (c) .idata and .edata, which encompasses respectively the list of imported and exported APIs for a dynamic linking library (DLL) or any other application (RAMAN et al., 2017; BINKLEY and EASTMAN, 2017).

In order to minimise the level of vulnerabilities in software security systems powered by low-level programming language, programmers require improved and reliable methods/techniques to conduct rigorous analysis of executable programs. Such methods will be developed to function at the binary data level given that this is the first point of entry to unleash an attack by a hacker. Across the years, few companies launched software products that can be adopted for vulnerability detection in binary executable files, however the methodological framework behind the software remains unpublished and test results were not rigorously verified (Tevis, 2005). As such, programmers have tended to use manual approaches or unproven methods for vulnerability detection in low level language programming.

The detection of vulnerability in binary executable files is not an easy proposition. To assist software practitioners in detecting vulnerabilities in binary executable program files, it is
important to develop robust techniques to address such challenges. The first requirement is to translate the executable program into an equivalent higher-level programming language using appropriate program transformation techniques. Details of how this is achieved in the context of the current work are provided in Chapter Three. In order to set the scene for which the vulnerability detection in binary executable files is carried, it is important to review all aspects of software vulnerabilities as highlighted in the section that follows.

2.2 Software Vulnerabilities

Software vulnerabilities are responsible for most of the security breaches reported in computer systems globally. Jimenez et al. (2011) defined software vulnerability as “weakness or an error in the system that can be exploited by an attacker in order to alter the normal behaviour of the system”. Another definition of software vulnerability as given by Liu et al. (2011): “the bugs in software, which may cause the violation of some security policies, may be used to tamper the confidentiality, integrity, and availability of the program or even the whole system wherein the program is deployed.” Similarly, Sotirov (2005) in his work defined software vulnerability as a phenomenon “that arise from deficiencies in the design of computer programs or mistakes in their implementation”. Given these definitions of software vulnerability, it is therefore unsurprising that a breach in software due to vulnerabilities can affect critical infrastructure including transport, healthcare, commerce and financial systems. Insecure coding practices, a shift in threat landscape, reuse of vulnerable components and code as well as the idiosyncrasies of programming languages are the four main sources of software vulnerabilities identified by Veracode (2013). More specifically, software vulnerability could emanate from error in programming, for example, misuse of unsafe function and features that are prone to error in programming languages including lack of array bounds checking and native string type as well as pointer arithmetic (Sotirov, 2005). A typical example of software vulnerability was highlighted in a flawed design for the Solaris, whereby a number of unauthorised users were able to gain access through the forgery of security credentials (Sotirov, 2005). As such, addressing vulnerabilities in computer software systems is therefore key to minimising threats that can cause severe damage and impact.

Based on the presentations above, it is established that attackers can easily exploit a computer that is vulnerable, compromising the overall system architecture, whilst allowing the attacker to gain access and take absolute control of the system with the view to damage it or orchestrate a new attack amongst other malicious intentions. Against this backdrop, it is pertinent to identify the various forms of vulnerabilities, their detection and prevention with the view to avoid or at
best minimise their occurrence in the final software version of the overall system. This will assist in reducing the possibility of attacks and damages that can be really expensive to mend.

2.2.1 Sources of Software Vulnerabilities

Software vulnerabilities arise in a number of ways and each way can impact the security posture of the software. It can occur due to error in the design framework of computer programs or inaccuracies during the implementation phase of the program. This could be as a result of misuse of unsafe and error-prone features of the underlying programming language employed in developing the software. (Veracode, 2013) identified four main sources of software vulnerabilities. First, insecure coding practice which entails the behavioural patterns, policies and habits during the software development phase which can lead to code vulnerabilities. Second, a shift in threat landscape whereby developers at the initial software development phase, adopt best practices and use resilient cryptographic algorithms, but at the final stage of the software production, the algorithm is broken. Such broken algorithms are no longer sufficient, but in most cases, the software development team is still unconsciously using it. Third, reuse of vulnerable components and code which entails the re-use of pre-built software components obtained from open source developers in order to accelerate the delivery of software development projects. Four, idiosyncrasies of programming languages whereby software development teams select the programming languages to adopt based on the type of application under development without taking into consideration the peculiarities of each programming language which makes them susceptible to various forms of vulnerabilities. As such addressing these sources of vulnerabilities is fundamental to minimising threats associated with computer software vulnerabilities.

Over the years, a great deal of effort has been placed on protecting computer networks and servers with little emphasis placed on security breaches and vulnerabilities resulting from low level programming. In part, this is due to the fact that most security professionals do not possess programming expertise for low-level languages. However, given today’s complex and highly competitive environment, security related to low level programming cannot be ignored. The creation of models or algorithms that can establish the set of conditions that could lead to or originate vulnerabilities is vital to addressing software vulnerabilities at the low level programming, and such understanding can then be utilized for prevention purposes (Jimenez et al., 2011). In the sub section that follows, an overview of various forms of software vulnerability, detailing their distinguishing features, especially as it pertains to low-level programming is presented.
2.2.2 Types of Vulnerabilities

In this section, a review of different types of software vulnerabilities is presented.

(a) Buffer overflows

This is one of the oldest forms of vulnerabilities in software and it’s arguably the most dangerous attack method used for exploiting computer systems (Christey, 2007; Ding and Yuan, 2011). It usually occurs with fixed length buffers when a program tries to insert more information or data in a buffer beyond the boundaries of capacity that is currently defined (Jimenez et al., 2011; Liu et al., 2011). A buffer in this situation is a sequential compartment of memory that is apportioned to hold any form of data including an array of integers or a string character. Writing out of range of the defined blocks of an apportioned memory could cause the system to malfunction, given that the introduction of new or foreign data can corrupt the data of other processes or buffers, causing the system to crash (Christey, 2007; Ding and Yuan, 2011). A typical example of buffer overflow is illustrated with C-programming syntax as indicated below:

```c
#include <stdio.h>
#include <string.h>
int main(void) {
    char buffer[14];
    int passNo = 0;
    printf("Please enter the password:\n");
    gets(buffer);
    if(strcmp(buffer, "helpmestuff"))
    {
        printf ("incorrect Password\n");
    }
    else
    {
        printf ("Correct Password\n");
        passNo = 1;
    }
    if(passNo)
    {
        printf ("\n***Root privileges given to the user***\n");
    }
    return 0;
}
```

Listing 2-1 Vulnerability Code in C Language

This lines of code illustrated above allows the user to gain access to the system by entering incorrect input with a bigger size and save it on the declared variable that has been specified at the beginning of the code. In this example, a variable call “buffer” was declared with size “14” but the command call “gets (buffer)” allows the user to enter any input with any size and save it on the variable that has been declared with size 14. This programming flaw can cause buffer
Chapter 2: Background and Literature Review

overflow on the stack memory whilst damaging the data on the neighbour memory. Accordingly, the command “strcmp” which means “string compare” will not recognize the fact that the declared variable is out of bound thereby setting the command “strcmp” into “confusion” while comparing with the password that was set to read "helpmestuff". The command “strcmp” will expect to compare with input size of maximum length 14 as specified and declared from the beginning of the code but will not recognize any data of bigger size while considering it as a similar value that has been compared with "helpmestuff". Based on this simple example, any form of buffer overflow or bug can be fully exploited by an attacker that is highly motivated.

(b) Format string bugs

Software vulnerability due to format string bugs are generally considered as a recent form of vulnerability given that they were first established to exist in 2000 after which a rapid discovery of vulnerabilities of similar patterns were reported in some software project development that are high-profile in nature such as FTP server, OpenBSD operating system to mention a few (Sotirov, 2005). Format string bugs are a somewhat simple form of vulnerability (Sotirov, 2005) that occurs when data from an external source is supplied to a functional output in the form of a format string argument. A typical example of format string bug is an output function within C programming language’s standard library termed printf, which generates an output based on the specification of in the format string’s directives. For example, if the printf function is used to print the username inserted in some fields of the page, the website could be vulnerable to this kind of attack. Some of these directives can write to memory locations specified within the format string, thereby allowing the attacker to gain control of the system through the exploitation of the printf to write malicious code into an arbitrary memory location, altering the control flow to execute it (Jimenez et al., 2011).

(c) Integer overflows

Integer overflows are another form of software vulnerability. They are much more difficult to exploit compared to buffer overflow and format string bugs. However, they are mostly found in Internet Explorer application, OpenSSH and in the kernel of Linux operating system (Jimenez et al., 2011). They occur when one tries to place an integer into the memory of a computer which is too large for the data type in a computer system. Integer overflows are of two forms namely: arithmetic overflows and sign conversion bugs. Whereas arithmetic overflow occurs when a value larger than the maximum integer size is recorded or stored in an integer variable, the sign conversion bugs occur when a signed integer is converted to an unsigned integer. Sign conversion bugs occurs on most new hardware platform when a negative number of small
Chapter 2: Background and Literature Review

magnitude is converted to an unsigned integer, turning it into a very large positive number. This can be illustrated with the lines of C codes shown below:

```c
1 char buffer [11];
2 int m = read_int ();
3 if (m < sizeof (buffer))
4 memcpy (buffer, src, m);
```

**Listing 2-2 Fragment of Vulnerability Code**

**(d) SQL injection**

This type of vulnerability occurs when codes are injected into a system with the view to exploit the content of a database. The attacker can then gain control and access to information that are considered sensitive in the database when the inputs are not handled correctly. The following line of code illustrates this vulnerability:

```
statement = "SELECT * FROM users WHERE name = " + userName + ";"
```

This SQL code is designed to pull up the records of the specified username from its table of users. However, if the "userName" variable is crafted in a specific way by a malicious user, the SQL statement may do more than the code author intended.

**(e) Cross site scripting (XSS)**

Cross site scripting (XSS) is a form of software vulnerability that mainly occurs in web-based applications through the injection of code into the web pages that other users have access to. The access controls can then be bypassed by an attacker to exploit this vulnerability by embarking on activities including identity theft, phishing, or expose connections. In the section that follows, a brief overview of probable causes of vulnerabilities is presented. The simplest way to show the importance of a XSS vulnerability would be to perform a Denial of Service attack. In some cases a Denial of Service attack can be performed on the server by doing the following:

```
article.php?title=<meta%20http-equiv="refresh"%20content="0;">
```

This makes a refresh request roughly about every .3 seconds to particular page. It then acts like an infinite loop of refresh requests, potentially bringing down the web and database server by flooding it with requests. The more browser sessions that are open, the more intense the attack becomes.

**2.2.3 Causes of Software Vulnerabilities**

Majority of the vulnerability types presented in the preceding section could be avoided to a certain extent if the software is designed and developed with a lot of care and attention, avoiding the introduction of vulnerabilities that an attacker can exploit. To minimise the damaging impact of software vulnerabilities, it is important for programmers and software developers to improve
their knowledge and understanding of known vulnerabilities in terms of their causes, threats, attacks and measures that can be employed to counter them. A number of models have been developed with the view to shape the understanding of software vulnerabilities and their causes. For instance, authors such as Byers et al. (2006) and Ardi et al. (2006) have developed models termed Vulnerability Cause Graph (VCG) for understanding the causes of vulnerability in software. The VCG is “a directed acyclic graph that contains one exit node representing the vulnerability being modelled, and any number of cause nodes, each of which represents a condition or event during software development that might contribute to the presence of the modelled vulnerability”. Specifically, Byers et al. (2006) used VCG to represent a known buffer overflow in xpdf (CVE-2005-3192) as depicted in Figure 2-1. As shown, the different probable causes and possible sequence of actions or scenarios that can lead to the occurrence of the kind of vulnerability under consideration. Essentially, the VCG concept can assist in gaining an understanding of the cause of the vulnerability so that instances are avoided during software life cycle development.

![Vulnerability Cause Graph (VCG), Adapted from Byers et al., (2006)](image)

Figure 2-1: Vulnerability Cause Graph (VCG), Adapted from Byers et al., (2006)

Now that the causes of software vulnerabilities have been identified, the next most important step in the literature review process is to review various form of techniques for preventing vulnerabilities in software as presented in the sections that follow.
Chapter 2: Background and Literature Review

2.3 Techniques for Detecting Software Vulnerabilities

Models and inspections as described in the preceding sections are useful to aid the understanding of vulnerabilities so that they can be easily prevented. However, it is also essential to consider tools, techniques or methods that can be adopted by software developers to enable them detect vulnerabilities when software is under development. Some of these tools are based on static methods (Puchkov and Shapchenko, 2005) whereby the source code of the program is analysed without necessarily running the code to perform the vulnerability detection. Static analysis is also used in reverse engineering of software systems and for facilitating the understanding of a program (Raman et al., 2017). Others are based on dynamic methods (Park et al., 2007) whereby a program is tested before it is run. The set of codes are executed in an environment that is highly controlled to conduct the detection or collect traces of program that can be used to actualise such aim. In contrast to static methods, dynamic method does not require a source code to be carried out but it can be inefficient due to lack of knowledge about the internal structure of the software under consideration (Ding and Yuan, 2011).

Another functional vulnerabilities detection mechanism which seeks to strike a balance between static analysis and dynamic testing through the integration of the attributes of both approaches have been reported to improve the security level of an application through the detection of more vulnerabilities and with greater precision (Ding and Yuan, 2011). A fine distinction between static and dynamic analysis is provided in Raman et al. (2017). In the subsection that follows a review of some static and dynamic techniques used for the detection of vulnerabilities is presented.

2.3.1 Static Techniques

As mentioned earlier, static techniques are those whereby the source codes are examined without running them with the view to evaluate or obtain precise information from the source code directly, in order to make judgements. Static techniques are commonly used for the optimisation of compilers. This is evident as most of the high-level optimisation carried out on modern day compiler was based on the results of static techniques including data-flow and control-flow analysis. Additionally, static analysis techniques have been used in various forms including the optimisation of software metrics, program understanding and refactoring as well as quality assurance (MA, 2017). In recent times, static analysis has been adopted as tools for code visualisation, uncovering more than a hundred of unknown bugs that were previously unknown in the kernel of Linux (Engler et al., 2011). When deployed for the detection of vulnerabilities,
static source analysis tool has the capability to identify specific vulnerable patterns of code or detect a possible path of control flow upon which a program attains a vulnerable state (Heffley and Meunier, 2016). As against dynamic techniques, static techniques have the capability of detecting vulnerabilities in code that are difficult to identify during the normal operation of a program (Puchkov and Shapchenko, 2005; MA, 2017).

Despite the numerous capabilities of static techniques highlighted above, they have some limitations. For instance, static techniques depend on a predetermined set of code patterns for vulnerability detection which are developed by highly experienced researchers in the field of vulnerability detection, thereby allowing users with minimal expertise in computer security to detect vulnerabilities that they would ordinarily not be able to discover given their limited expertise. However, there is a limitation based on the fact that such tools are only able to identity and report only vulnerabilities that is captured within their rule set. As such, the fact that such tools report no vulnerabilities in a program does not validate or confirm that vulnerabilities didn’t actually exist. In instances that a given rule within a static source analysis framework is written to identify a particular vulnerability, it is still possible that not all vulnerabilities will be detected.

Also, in static analysis, it is generally expected that vulnerability detection must be performed at a relatively fast pace, but this normally happens at the expense of approximations. Such approximations may result in false negatives and false positives as part of the output of the tool. The number of false negatives, for example, can be reduced by making the analysis to be more conservative and restrictive and flag potential vulnerabilities. This has the tendency of increasing the number of false positives beyond a level that is manageable. At the same time, rendering the analysis less conservative have the potential of lowering the number of false positives but some vulnerability might still be left undetected. This type of conundrum can be addressed by developing static tools with the capability to ensure that the number of false positives as low as possible. A review of the various forms of techniques used to perform static analysis, detailing their merits and demerits is discussed in the succeeding paragraphs below.

(a) Pattern Matching

Pattern matching is the simplest form of static analysis technique and it entails searching for a “pattern” of strings within the source code of a program to ascertain the number of occurrences of any form of vulnerabilities. In C programming language, for example, such pattern could be any a vulnerable function such as “gete”. The implementation of pattern matching can be achieved through the use of simple UNIX command such as “grep” but such approach have the tendency to generate a lot of false positives due to lack of analysis of the results (Jimenez et al.,
Chapter 2: Background and Literature Review

2011). Furthermore, its effectiveness is limited in scope given that it depends on the exact manner with which strings are written, creating extra white spaces that can limit the results. Within a C programming language, pattern matching tool lack the capability to differentiate between comments from real code and can deceive unexpected whitespace and macros due to unavailability of proper C parser (Sotirov, 2005). Hofer (2010) demonstrated the use of flaw finder as an elaborated pattern matching process to identify potential vulnerabilities, sorting them based on their level of risk. Such risk levels rely entirely on the function as well as the values of the parameters of the function.

(b) Lexical Analysis

Lexical analysis imposes an extra layer of improvement before the application of a pattern match. A lexer can be adopted to transform a source code into a sequence or stream of tokens, removing whitespace, which are then compared against a vulnerability database so as to identify them. Lexical analysis has the capability to improve the accuracy and efficiency of pattern matching because it can handle irregular code formatting and whitespace (Jimenez et al., 2011). However, the benefits of lexical analysis are limited given that false positive number reported by it is still high due to lack of consideration given to syntax or grammar of the source code. The technique based on lexical analysis was adopted in popular tools such as Flawfinder (Wheeler, 2007), RATS (rough auditing tool for security) and the ITS4.

(c) Parsing

Parsing of a source code is another technique used for the improvements of the accuracy of static analysis techniques. Compared to lexical analysis, parsing is characterised by some level of complexity. Accordingly, when the source code is subjected to parsing, a representation of the program is constructed based on a parsing tree (i.e. an abstract syntax tree, AST) in order to analyse the semantics and syntax of the source code (Jimenez et al., 2011). This activity is usually carried out using the frontend of a compiler, offering a great opportunity for the reuse of code. Su and Wassermann (2006) employed the parsing technique to detect SQL command injection attacks on a web application. In high level programming languages such as C and C++, the adoption of parsing technique on their source code is a very tedious task to achieve. This is because most compilers share compatibility with C standards including C99 and ANSI C but most programs depends largely on at least one or more of non-standard language attributes, bugs or compiler extensions (MA, 2017).

(d) Type Qualifier

Type qualifier framework was developed by Foster (IGWE, 2017) as a technique for vulnerability detection. They are adopted to qualify the standard types and for the modification
of the properties of variables in the chosen programming language. This was the basis of the CQUAL tool ("a lightweight type-based analysis engine for C programs") which was developed to specify and check C programs properties based on type qualifiers that are user-defined and injected into the program (CQUAL, 2004). The modified version of the program is then analysed for the detection of vulnerabilities. Shankar et al. (2017) adopted the type qualifier technique for format string detection system.

(e) Dataflow Analysis

The goal of data flow analysis is for the determination of the possible value which a variable or an expression must possess while a program is being executed. It is mostly used for detecting buffer overflows, array indexing, integer overflow as well as pointer arithmetic problems which techniques such as pattern matching cannot handle. The aforementioned vulnerabilities come into play when variables within a source code can accept values outside a definite safe range. For instance, the function call strcpy () can become vulnerable if the source string size surpasses the destination buffer size. As such, the detection of the circumstances under which this type of problem can happen without running the program can constitute a difficult problem, hence the advantage of dataflow analysis which has the capability to address such problems (Sotirov, 2005). Kim et al. (2008) adopted dataflow analysis by using rules which describe vulnerability pattern and the source code for the detection of locations and patterns in the program. The processes were executed based on three steps namely pattern matching, control and data flow and flow analyser.

(f) Taint Analysis

Taint analysis is a concept of tainting and is a distinct form of data flow analysis whereby any form of data emanating from sources that are not trustworthy (e.g. introduced by an attacker or a network), and which constitute a threat to the system, are marked as “tainted”. Usually, within a software system, a security policy which specifies which untrusted data are restricted or permitted are defined and any attempt to use any form of tainted data in an attempt to violate such policy is considered a form of vulnerability. Accordingly, tainted data flow is monitored and prevented to reach functions that are adjudged critical unless it is processed and becomes untainted. For instance, Livshits and Lam (2005) developed a framework based on static analysis to detect vulnerabilities in Java applications, where they defined a class called Tainted Object Propagation problem, to address problems with inadequate user input validation. Vulnerability specifications bytecode written in Java were employed to perform taint object propagation and detect vulnerabilities using the Eclipse platform. Similarly, taint mode technique is provided by Perl programming language (IGWE, 2017).
(g) Model Checking

Model checking automatically tests whether the underlying model upon which software is constructed meets its required specification and can then be used for the detection of vulnerabilities. Model checking is usually a complex approach due to the difficulty in elaborating on the model but once such models are obtained, it becomes relatively easy to test the properties of the system. Hadjidj et al. (2008) developed a security verification tool within a multiple language support framework based on GCC compiler. In their framework, an orthodox push down system model checker was employed to verify the security properties of a software based on three sequential steps including specifications of security property, extraction of program model and the checking of the properties of the model. Their output was able to detect errors with execution traces. Similarly, Wang et al. (2008) integrated model checking with constraint analysis to detect vulnerabilities due to buffer overflow. In their work, memory size of buffer-related variables was traced and the codes were then engineered with constrains assertions before the potential vulnerable points which was then used to detect vulnerabilities.

Numerous researchers have employed static analysis for the detection of vulnerabilities. Table 1 to 3 in Appendix A gives a summary of past work pertaining to buffer overflow (BOF) vulnerability detection; classification of the analysis based on BOF vulnerability detection, as well as comparison of performance analysis of vulnerability detection mechanism. The above section detailed static techniques for detecting software vulnerabilities, including various forms and their merits and demerits. In the section that follows, a review of dynamic techniques for vulnerability detection is presented.

2.3.2 Dynamic Techniques

To detect vulnerabilities using dynamic techniques, it is essential to run the program or source code, and then analyse the behaviour of the system with the view to generate a decision. A review of some of the techniques for carrying out dynamic detection is presented below:

(a) Fault Injection

Fault injection as the name suggests involves a technique whereby faults are introduced into a system with the view to test the behaviour of the system. To embark upon a successful fault injection routine, knowledge of the system under consideration is required to generate and identify the possible faults. YAQOOB (2017) employed fault injection to conduct vulnerability checks for software system using to find security flaws. In the study, faults were injected into the system under test followed by an observation of the behaviour of the system. Final deductions were based on the fact that if a system fails to accept faults, then it is an indication of a flaw in
the security settings of the system. The propagation of a fault through to an observable failure follows a well-defined cycle. When executed, a fault may cause an error, which is an invalid state within a system boundary. An error may cause further errors within the system boundary, therefore each new error acts as a fault, or it may propagate to the system boundary and be observable. When error states are observed at the system boundary they are termed failures.

(b) Dynamic Taint

Dynamic taint is similar to taint analysis with the main difference being that the tainted data is observed during program execution to ascertain its validity before functions that are sensitive are entered. Based on this technique, it was possible to discover potential problems that pertain to input validation which are identified and described as vulnerabilities (Chess and West, 2008).

(c) Fuzzing Testing

Fuzzing testing is a technique whereby a random data is used as an input to software to ascertain if the software can manage such input correctly. The implementation of fuzzing testing is easier as compared to fault injection given its simpler test design and the fact that a prior knowledge of the system under consideration is not always a requirement and the test are limited to the point of entry of the program (Jimenez et al., 2011). An example of fuzzing testing can be found in web scanners. Fuzzy testing can be improved upon to make the coverage of the system better. For instance, McAllister et al. (2008) in their work on which focus on the expansion of human interactions for detailed web applications testing recorded real user inputs filled out on web forms. The data collected were then used for fuzzing testing process with the view to better explore web applications in terms of wider reachability.

(d) Sanitization

One of the probable ways to prevent vulnerabilities that emanates from data supplied by a user is to implement novel incorporated functions or routines that are customised for the validation or sanitisation of any form of input from the users prior to usage in a program. For instance, Balzarotti et al. (2008) presented a technique based on static and dynamic analysis to determine the level of correctness of the sanitisation process in web-based applications that an attacker could easily bypass. They adopted data flow methods to find the flows of input values from places where the value is used. Dynamic analysis was then applied to determine the correct sanitization process.

The preceding sections have focused extensively on the subject of software vulnerabilities, laying the foundations for this research. As observed, vulnerability in software is not a new topic in the field of software development but they consistently appear in programs or source codes suggesting that the need to minimise vulnerabilities in software cannot be overemphasised. To
develop better and improved software, programmers must understand the concept of vulnerabilities in software and try to avoid their presence as much as possible. The selection of the vulnerability detection to use is based on the type of application that is under test and evaluation, the programming language used and the type of the vulnerabilities to be detected. As stated in chapter one, the focus of the current research is on vulnerability detection in legacy systems powered by assembly language. As such, in the section that follows, a review of past works focusing on vulnerability detection with a focus on buffer overflow is presented. This is to allow for a fine distinction to be drawn between what the current research seeks to address and what other researchers in the field have addressed so far. This forms the basis of the gap in knowledge which the current work addressed.

2.3.3 Overview of Binary Code within Vulnerability Detection Methods

In the computer software parlance, attacks emanating due to buffer overflow in binary executable files of assembly language is regarded as the commonest and the most dangerous form of computer software vulnerabilities and their tendency to continue to be a source of major concern for a long time in the software development community is very high. As such, a great deal of work has been developed with the view to minimise the overall threat of vulnerabilities in binary executable files. For instance, Ding and Yuan (2011) presented a method which integrates static analysis techniques with dynamic test to identify vulnerabilities due to buffer overflow based on binary code and demonstrated its feasibility. They showed that the integration of the two approaches of static analysis techniques and dynamic testing can complement each other, playing important role in identifying software vulnerabilities. In their work, Liu et al. (2011) developed a novel framework that detects Vulnerability of Flaw Function Misusing (MFFV), using the reverse engineering technique, with a demonstration of the feasibility of the model. Similarly, Jimenez et al. (2011) employed existing models (e.g. Cavalli et al., 2008) for the development of tool-based for automatic detection of vulnerabilities low level programming.

Brumley (2008) developed techniques for vulnerability analysis and defence that only require access to vulnerable programs in binary form, based on an approach that does not entail the use source code. Adopting a binary-centric approach, a demonstration of the capabilities of work to a wider audience as compared to previous approaches that require source code was achieved. Newsome and Song (2005), proposed a dynamic taint analysis technique called “TaintCheck”, for automatic detection of overwrite attacks, including many forms of exploits. Their approach did not require source code or special compilation for the program under monitoring and was able to reliably detect most types of exploits, producing no false positives for many of the
Chapter 2: Background and Literature Review

various programs tested. Also, Toth and Kruegel (2002) developed an approach that accurately detects buffer overflow code in web applications. Dor et al. (2003) proposed C String Static Verifier (CSSV), a tool that has the capability to identify all string manipulation errors. The CSSV prototype was used to ascertain the absence of errors in real code from EADS Airbus and many other string intensive applications.

Tevis (2005) developed a model for automatic detection of software security vulnerabilities in executable binary files using information located in the headers, sections, and tables of a Windows NT/XP executable file alongside the information derived from the entire contents of the file without the need to disassemble the code. The model was validated and verified through the test results on 2700 files where it was shown that the model was able to detect a number of anomaly including large zero-filled regions of bytes, table size anomalies, unrecognisable regions of bytes, the use of functions that are susceptible to buffer overflow attacks etc. BINKLEY and EASTMAN, (2017) provided detailed explanation regarding the application of conventional intraprocedural static analysis to binary executables for static analysis of machine-code and assembly code using program slicing techniques. This entails debugging the code and determining the instructions that affect an indexed jump or an indirect call on a register. Their work was shown to be useful in the decoding of machine instructions phase of reverse engineering tools of binary executable files (e.g. binary translators, profilers and debuggers).

RAMAN et al. (2001) proposed a novel technique for the static detection of malicious codes in executable programs using semantic analysis based on behavioural patterns that also allow for the possible detection of malicious codes that were previously unknown. The overall analysis was carried out directly on binary code using static analysis of binary executable files based on three main steps including construction of an intermediate representation, flow-based analysis that accounts for security-based program behaviour and static verification of critical behaviours against security policies. In their work, Rawat and Mounier (2012) developed techniques for detecting Buffer Overflow Inducing Loops (BOIL) in binary executable files. Their model, though simple but efficient shows that vulnerability pattern was effective in practice for vulnerability detection in binary executables, yielding a major reduction on the size of codes to be analysed. The development of automated vulnerability detection models for binary executable files for real-life practical applications is an important yet difficult challenge. As such all the aforementioned researchers have contributed in one way or the other to address the challenges of vulnerability detection in binary executable files. However, most of the methods employed by these researchers are based on informal procedures which can greatly affect the accuracy of the vulnerability detection exercise. Against this backdrop, the current work seeks to explore the
feasibility of employing formal methods such as the use of FermaT Transformation Engine within a Wide Spectrum Language (WSL) framework for vulnerability detection in binary executable files. For high-level programming languages, the use of transformation techniques within WSL framework have been touted to provide better and improved capability in handling vulnerability detection in a formal way. In terms of their application for vulnerability detection in low-level programming language, little research has been conducted to explore their viability. Against this backdrop, the gap which the current research seeks to close is to investigate and identify potential transformation methods within WSL which can be used for vulnerability detection in binary executable files. In doing so, it is intended that a robust and efficient mechanism based on formal methods supported by proven mathematical algorithm will assist in shedding light on the challenges of vulnerability detection in binary executable files.

To be able to fill the gap identified above, certain key concepts and techniques will be required especially as it pertains to software re-engineering and software maintenance, which seeks to highlight the processes involved in improving or transforming existing software. To this end, it is important to review these aspects of the current work as detailed in the sections that follows.

2.4 Techniques for Preventing Software Vulnerabilities

Models such as VCG discussed in section 2.2.3 constitute the first most important step to tackle vulnerabilities given that it provides a basis for understanding them. However, it is equally pertinent to understand methods or procedures or mechanism for preventing the risks associated with software vulnerability. Accordingly, in the section that follows two possible vulnerability prevention methods (i.e. software inspection and security graph activity) as established in the extant literature are presented. The purpose of understanding software vulnerability prevention techniques is to allow for the evaluation of the code during the development phase of the software to identify any security defect and nullify them within time thereby reducing the need for conducting intensive debugging and test at the end phase of the software development when the entire software operational themes is completed. A brief description of techniques for the prevention of software vulnerabilities is provided below:

2.4.1 Software Inspection

Software inspection is a process that entails the reading or visual inspection of the program code or documentation with the view to detect any form of defects and correct them at the early phase of the software development. This is important because if the defect is established on
time, it become less expensive to fix. However, the effectiveness and efficiency of a very good inspection relies entirely on the technical expertise and prowess of the software inspector and the type of defects that is expected to be identified. A brief overview of two methods of inspection for the classification of the implicit knowledge of vulnerability prevention specialists as it pertains to the checking of the accurate implementation of security strategies and how to investigate vulnerabilities.

(a) Security Goal Indicator Trees

Security Goal Indicator Trees (SGIT) – attention is focused on the features of software that are regarded as positive and can be easily verified during process of inspection (Peine et al., 2008). It is a graphical structure whereby the root represents a security objective with its subtree representing properties or indicators that can be verified to achieve the security goal. Given that not all features or properties can be expressed in a positive manner, it is also possible to have indicators or properties that are negative (i.e. an event that shouldn’t occur). These indicators are constructed based on Boolean relations to realise the objective and must be checked in order to ascertain the security objective. SGIT are specifically designed by software security experts. For instance, Peine et al. (2008) developed a SGIT for Audit Data Generation (Figure 2-2) highlighting some dependency relations as well as indicators that are both positive and negative.
(b) Vulnerability Inspection Diagram

Another type of manual inspection method is known as Vulnerability Inspection Diagram (VID) which was developed and introduced by Sheffield Project Consortium (2008) to allow software developers to leverage the expertise and experience of computer security experts as it pertains to the problem detection in the lifecycle of the software development. Essentially, a VID is simply a graphical illustration in the form of a flowchart which serves as a guidance or protocols for software developers towards detecting the existence of vulnerabilities based on the knowledge of the software expert.

2.4.2 Security Activity Graph

Security Activity Graphs (SAGs), which are tools that depicts causes in a VCG in a graphical form was employed by Ardi et al., (2006) and Byers and Shahmehri (2008) as a means for the prevention of software vulnerabilities. SAGs provide an indication of how a given problem can be prevented by adopting a combination of security strategies during the development phase of a software. This is illustrated in Figure 2-3 showing various alternatives as to how a cause termed “Lacking design to implementation traceability” could be addressed.

![Security Activity Graph](image)

**Figure 2-3: Security Activity Graph, SAG, Adapted from Byers and Shahmehri (2008)**

The preceding section describes various ways of preventing software vulnerabilities during the development phase of software. In the section that follows a review of various mechanisms for detecting software vulnerabilities is presented.
2.5 Software Re-Engineering and Software Maintenance

Generally speaking, software re-engineering entails taking a current legacy software whose maintenance cost has become unbearable and whose system architecture or mechanism are obsolete with the view to redesign it with modern software and/or hardware system (RASHID, N. and KHAN, 2013). The level of difficulty of re-engineering a legacy system lies in the understanding of such systems given that requirements, design and documentation of source codes are no longer available or in some cases, completely obsolete, so that it becomes uncertain what functions or attributes are to be removed (RASHID, N. and KHAN, 2013; ARIKPO, 2017). On the other hand, software maintenance is a form of software re-engineering which involves the modification of software to improve its performance. Against this backdrop, a review of software re-engineering and software maintenance in terms of why it is important to embark upon it and its underlying objectives is presented in this section.

RASHID, N. and KHAN (2013) described software re-engineering as a process “to improve or transform existing software so that it can be understood, controlled, and used anew.” In their work titled: “Reverse Engineering and Design Recovery: A Taxonomy”. GERSTMAYER (2017) defined software re-engineering as “an examination and alteration of a system to reconstitute it in a new form”. In simple terms, software re-engineering is the modification of a software system by adding new functions with the view to correct errors or flaws. The American National Standards Institute (ANSI) and the Institute of Electrical and Electronics Engineers (IEEE) (ANSI/IEEE Std 729-1983) defined software maintenance as “the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment”. The importance of software re-engineering has increased tremendously, given that legacy software systems have become outdated in terms of their systems architecture, the platforms upon which they operate and their appropriateness to support the evolution of the software under consideration (RASHID, N. and KHAN, 2013; Ryan, 2000). Software re-engineering is therefore useful in the recovering and reusing of existing software assets of a system, minimising the cost of software maintenance, and establishing a base for the evolution of future software (RASHID, N. and KHAN, 2013).

Currently, the number of legacy systems that are in use today are very high but their functionality remains constant while in modern systems, the application environment, software and hardware platforms are different, making interoperability very difficult. Essentially, to improve interoperability, there is a need for re-engineering of the legacy system. However, most legacy systems are severely lacking in terms of efficient design structure and source code
organisation, rendering changes to such systems difficult and expensive, yet it is important to re-engineer such systems to be in line with modern system. The overall objective is to gain an understanding of the existing software in terms of design, specification and implementation to achieve an advanced level of abstraction (GERSTMAYER, 2017). Essentially the overall benefit or objectives of software re-engineering is to: (i) increase the maintainability and functionality of the system; (ii) achieve performance improvements; (iii) increase the interoperability between systems (i.e. seamless migration); (iv) decrease the personal dependency on low-level software engineers (i.e. improvement in reliability) (GERSTMAYER, 2017; RASHID, N. and KHAN, 2013).

2.5.1 Reverse Engineering

Reverse engineering are steps carried out with the view to improve software products as well as to analyse software systems. The use of this technique not only assist in the basic understanding of the underlying system and its basis structure, but also helps to understand the system’s design level, aid maintenance, strengthen enhancement or support replacement (GERSTMAYER, 2017). When embarking on reverse engineering, the design requirements, system structure and internal content of the system under consideration must be recaptured (RASHID, N. and KHAN, 2013). The key objectives in reverse engineering include (GERSTMAYER, 2017; RASHID, N. and KHAN, 2013): (i) identification of key components of the system and their interrelationship; (ii) development of a high-level abstraction representation of the existing system; (iii) creation of different views, recovering of missing data or information, identification of downsides, and facilitation of reuse. In reverse engineering, it is not mandatory that the structure of an existing system be changed and neither does it require that a new system be created. It entails a detailed examination of a system whilst maintaining the overall integrity and functionality of such system. Figure 2-4 shows a representation of the reverse engineering process. As shown, the process of reverse engineering commences by gathering the overall requirements, design and detailed structure from existing documents related to the system under consideration and the source code. This is then followed by the formulation of a document detailing the requirements based on high level abstractions and design using tools such as control and data flow diagrams. The penultimate step is checking for correctness and consistency of the re-engineered system (RASHID, N. and KHAN, 2013).
Figure 2-4: Reverse Engineering Procedure.

There are two sub-areas within reverse engineering procedure namely: design recovery and re-documentation (GERSTMAYER, 2017). Design recovery is another variant of reverse-engineering, which assist in the identification of meaningful higher level abstractions of the system, by adding information derived externally, domain knowledge, and reasoning deduced based on fuzzy logic. Design recovery must be consequently reproduced with the help of knowledge, which includes the required and adequate information for a system operator to fully comprehend what the program or system does. Re-documentation entails the formulation or modification of a semantically equivalent representation within the same level of abstraction. They can be alternative views of data structure and control. It is known as the oldest form of reverse engineering. Their key feature is to realise and visualise the relationships among program components and their interrelationship.

2.5.2 Forward Engineering

This is the traditional procedure for migrating downwards through the levels of abstraction from abstractions based on high level routines and logical implementation and design to the practical implementation of the system (GERSTMAYER, 2017; RASHID, N. and KHAN, 2013). In actual sense, the downward movement is a form of forward movement based on standard process in software development, hence it is termed forward engineering. It is based on
Chapter 2: Background and Literature Review

a sequence of processing steps, starting from the requirements level through to designing and implementation. In forward engineering, systems are open to more risk during the modification or inclusion of new systems requirements (RASHID, N. and KHAN, 2013).

2.5.3 Restructuring

Restructuring entails the conversion from one form of representation to another based on similar abstraction level, while maintaining the external behaviour (i.e. semantics and functionality) of the system under consideration and improving its internal structure. An example can be found in FermaT transformation system (Ward, 1999). A restructuring transformation entails the alteration of code in order to improve its structure, as in the case of a code to code transformation which recasts a program from an unstructured “spaghetti source code form” to a structured form. Restructuring processes can be performed without having the knowledge of the program behaviour. This opens the opportunity, to case a set of “if statements” into a “case structure” or vice versa, without knowing the program’s purpose or anything to do with the problem domain. Therefore, restructuring can be regarded as a creation of new versions of the code without modification to what it does not include in the new system’s requirement and it may lead to a better understanding of the system under consideration for future code adjustments and software maintenance. Figure 2-5 shows the overall linkages between all forms of software re-engineering.

![Figure 2-5: Overall Pictorial Representation of Software Re-engineering.](image)

2.6 Program Transformations

In the last 30 years, methodological framework of software based on program transformation has been ascertained to be a powerful and useful mechanism for deriving programs from specifications, verification of program properties and specialisation of programs especially as it
Chapter 2: Background and Literature Review

pertains to the context to which they are used and the derivation of more efficient and improved program versions from less efficient ones. NARASIMHAN and REICHENBACH (2017) were the first to propose a robust methodology for transformation in the field of functional programming. Thereafter, Tamaki (2011) applied the concept of transformation methodology in logic programming and many other authors including Kawamura and Kanamori (2014); Gardner and Shepherdson (2008); Seki (2011); Shepherdson (2017); Sato (2010); Seki (2012); Aravindan and Minh (2013); Pettorossi and Proietti (2014) in their respective research efforts have all developed many approaches of program transformation methodology in area of functional and logic programming as well as in other programming paradigms.

The concept of transformational programming is a constructive approach that involves the successive application of a number of transformational rules through the generation of a new program from an original program whilst ensuring that semantics are unchanged (WANG, 2013). Transformations are more powerful in providing flexibility towards implementing sets as lists, arrays, hash tables and etc. The transformation mechanism is initiated through the specification of program which is the successively transformed into an executable program through series of transformations that have been established to maintain the absolute correctness of the program. Essentially, a transformation exercise is said to be correct if the semantics are equivalent after the transformation. For instance, NARASIMHAN and REICHENBACH (2017) in their work submitted that “programs are complicated, hard to understand and prone to errors, because we want them to be efficient. So the idea is to start with a programme which does the right job, but entirely sacrifices efficiency in favour of simplicity and modularity. We then transform it by correctness-preserving transformations, until a tolerably efficient, though less perspicuous, programme is obtained.” Accordingly, two conditions must be met before a transformation programme could be adjudged successful: (i) ensure that specifications that are capable of being transformed into an efficient program are produced and (ii) ensure the possibility of transforming an existing programme into a specification (NARASIMHAN and REICHENBACH, 2017).

Transformation systems can be broadly categorised into two categories based on their purposes: (i) those used for specific but limited purposes and (ii) those employed for general purpose manipulations in programs used as tools. Whereas those meant for specific purpose are executed by super compilers - automated transformational programming systems, which have the capability for the translation of high-level mathematical problem specifications into codes in machine language equivalent for the need of wide range of computers (Partsch, 2014); the special purpose transformations find useful applications in software maintenance and also in reverse
Chapter 2: Background and Literature Review

engineering based on powerful transformation techniques. Overall, program transformation has a number of advantages including (KLUGE, 2017) the transformation system with sound theoretical foundation which can produce a program that can be relied upon to be correct by construction; semantic rules based on transformations can be adopted for an entire class of situations and problems; total program process and development will be supported by computers due to formality (includes the changes made to code by hand; probability of finding errors in this method are more); and program structure is flexible as it is not fixed throughout development process. These advantages find applications in software maintenance as they are mainly related to the fact that they are produced formally as the program produced is from program specifications. Other advantages include (Yang, 2003) increase in reliability, errors and inconsistencies are detected easily at high level abstractions. Maintenance can be embarked upon at specification level and formal links can be maintained. Programs can be improved incrementally and data structures and abstract data types are changes easily.

2.6.1 Program Slicing

In section 2.3, a number of software vulnerabilities detection techniques under the two broad categories of static and dynamic techniques were highlighted and discussed. However, tools for detecting vulnerabilities in binary executable files especially as it pertains to buffer overflow still suffer a number of limitations due to the difficulty and time-consuming nature of such endeavours. To detect vulnerability in a binary executable program in the absence of the source code, it can take even longer time to achieve given the manual approach that programmers have adopted in the past (BINKLEY and EASTMAN, 2017). To reduce the amount of time and efforts for vulnerability detection in binary executable files, the use of efficient and reliable automated tools will be required. In the past, reverse engineering tools such as static binary translators (Sites et al., 2015) for vulnerability detection in binary executables files has been developed. Although such translators offer a rapid approach towards the translation or migration of binary executable files from one platform to another, they ultimately relied on runtime interpreter for the translation of any code that wasn’t translated statically (BINKLEY and EASTMAN, 2017). The use of slicing techniques can be used to improve the speed and reliability of vulnerability detection in binary executable files.

Against this backdrop, a number of authors including Weiser (1981); Horwitz et al. (2016); Gallagher and Lyle (2011); BINKLEY and EASTMAN, (2017); Zhang (2007) and many more have adopted the technique of concept program slicing to improve the speed of vulnerability detection. Program slicing was originally introduced by Weiser (1981) and has been widely
adopted in software maintenance, program debugging, code analysis, software testing, reverse engineering and many more applications. It is a technique adopted for the reduction of the volume of information that is required by a programmer in exploring a software program. Given a specific point of focus within multiple lines of codes described by a variable and a statement, the use of program slicing techniques can be used to highlight all the statements that contributed to the value of the variable at the given point whilst disregarding statements that are deemed unnecessary (Binkley and Gallagher, 2007). For instance, during the process of debugging a program if it is desired to identify the line of codes causing the program error within a very large program containing several lines of codes, then program slicing can be adopted to exempt the codes that have nothing to do with the program error, thereby limiting the error identification to a focused and smaller range. This has the advantage of decreasing program debugging time. Program slicing could either be based on static slicing, dynamic slicing, forward slicing, backward slicing, single routine slicing, multiple routine slicing, distributed and non-distributed slicing. A detailed review of program slicing techniques and their respective applications under different programming scenario is provided by Tip (2010) and Binkley and Gallagher, (2007).

2.6.2 Static Single Assignment

Static Single Assignment (SSA) entails the structuring of an intermediate representation (IR) in such a way that each variable is assigned only once. Developed by IBM, SSA allows use-def. chains to be structured in an explicit manner in the IR, thereby allowing for the simplification of program optimisation to get rid of computational procedures that are adjudged redundant. SSA allows for a representation that is compact and in terms of updating, it is much easier depending on the optimisation task at hand. Within a program environment, SSA form makes arrangement for every value calculated by a program to possess a unique definition and attribute. SSA can be used to eliminate redundancy such as value numbering, conditional constant propagation as well as common subexpression. Modern day compilers are endowed with the capability to transform a program into an SSA form before program optimisation takes place. The SSA form possess the attributes that each variable in the program is the target of only one assignment. Transformation is carried out through the creation of various versions of a variable, each representing an assignment. The SSA form of a program gives room for optimisation to be carried out in an efficient manner, simplifying the implementation procedure in the process. A demonstration of the application of SSA is provided by Hasti and Horwitz (2016).
Chapter 2: Background and Literature Review

2.7 FermaT Transformation Engine

The FermaT transformation system is a powerful and dominant program transformation system based on a Wide Spectrum Language (WSL) language (Ward, 2004). FermaT focuses lies on re-engineering IBM Assembler 360 mainframe code. It has been successfully used in a number of migration projects that involves the transformation of millions of lines of assembler codes written based on manual procedures to high level programming language like COBOL or C that are more efficient and easily managed. It has also been used to translate modules written in assembly language to intermediate programming language such as WSL. Essentially, FermaT is used to simply, restructure, maintain or raise the abstraction level of program code, by utilising WSL program transformations. Within each processing step, the maintainer usually has to evaluate on which path a transformation should be applied. Usually after the application of a sequence of transformations, it becomes complicated to evaluate the result, because it has to be based on personal knowledge. This is also the case if more than one transformation sequence is applied on the same program source or state. The difficulty also lies in the enormous applicability of code transformations which commonly slows down the transformation process. A transformation process pre-processing technique could eliminate this problem. An analysing system which evaluates applicable transformation sequences together with reliable transformation search tactics could automate and speed up the whole transformation process.

2.7.1 The Wide Spectrum Language (WSL)

Wide Spectrum Language (WSL) has been developed for almost two decades and has been employed in a number software re-engineering and reverse engineering activities (Ward, 2005). The WSL is so labelled because of its inherent ability to utilise a wide range of mathematical specifications for program transformation (Yang and Ward, 2003). It contains both specification constructs (e.g. general assignment statement) and programming constructs (e.g. while-do loops) (Younger and Ward, 1993). Given these unique attributes of WSL, it is generally employed for re-documentation and re-engineering purposes. By translating the source code of a legacy system to an intermediate representation within a WSL framework, a tool for conducting re-engineering can be derived to generate new documentations for the system under consideration (Yang and Ward, 2003). The use of WSL as an intermediate language or representation has many advantages such as the inherent ability to adopt standardised mappings and transformations from the intermediate to the target domain thereby avoiding type-mismatch between the target
Chapter 2: Background and Literature Review

domain and the source. This allows the re-engineering effort to be subdivided into smaller steps as against a monolithic source to target domain re-engineering effort (Milicev, 2002).

WSL was developed with several advantages in mind: (i) it has simple, regular and formally defined semantics; (ii) its syntax is simple, clear and unambiguous; (iii) WSL has the capability to represent general specifications in terms of logic with mathematical basis using suitable notation; (iv) WSL is supported by advanced library functionalities with proven transformations capabilities, requiring minimal efforts from the user when dealing with complex proofs and transformation challenges; (v) WSL possesses chains of functions, constants and procedures which are especially in conformity with WSL programs. This collection of functions and procedures is collectively known as MetaWSL which is extended from WSL for writing program transformations; (vi) Techniques based on WSL can close the gap in terms of abstraction between specifications and implementation; (vii) WSL has the ability to scale up large programs with a wide range of industrial applications; (viii) WSL has existing tool support as well.

It was highlighted in section 2.3.3 that a number of models/framework for vulnerability detection in binary executable files have been developed using informal methods. In the current work, the use of formal methods which leverages on the proven ability of WSL to transform programs as well as its capacity to represent high and low levels of abstraction for program re-engineering purposes is investigated. A demonstration of such viability of WSL through the extension of its program transformation engines to allow for vulnerability detection in binary executable files of low-level language such as assembly language is the hallmark of the current work and it will hopefully deliver the overall novelty of the research.

2.8 Difference between the Proposed Approach and the Signature Technique used in Anti-Viruses.

At the face value of it, signature technique used in anti-viruses and the technique reported in this work may appear to share some similarities in principle however there is marked difference between the two concepts. In signature-based detection, key aspects of a given file is employed to establish a static fingerprint of any form of malware and the signature may be represented as a series of bytes within the file or as a cryptographic hash of the file (Ref here). Since the inception of the concept of anti-virus, signature technique has become an integral part of malware detection in antivirus kits. Although it recorded significant level of success but their relevance is gradually diminishing due to the inherent limitations that comes with their adoption. For instance, the use of signature-based detection mechanism is unable to detect malicious files whose signatures have not been developed and attackers leverage this vulnerability by mutating
Chapter 2: Background and Literature Review

their codes to retain malicious functionalities which can alter the signature of the file. This is particularly the case given that antivirus techniques are based on a dynamic analysis based on static signatures which can be different from those signatures that are already predetermined within the database of the antivirus. Essentially, if there is a mismatch between the predetermined signature in the database and a new form of malicious code, the detection become obscure to the antivirus. Accordingly, the signature technique used in anti-viruses for detection of vulnerabilities is no longer as potent as it used to be prompting the development of new techniques such as the one developed in this thesis.

The approach reported in this thesis is based on reverse engineering technique which detect the vulnerabilities by pointing out the exact line in the code thereby helping the developer to easily detect the flaw and address it accordingly. Vulnerabilities in software can be viewed as errors unknowingly inflicted by the developers or problems which emerges due to errors such as no checks on stack boundaries etc. As highlighted in the preceding sections of this chapter, vulnerability issues are fundamental to curbing a lot of problems in the computer parlance, as such new techniques as demonstrated in this thesis are pertinent. In the technique developed in this thesis, source codes are analysed thoroughly with the overall aim of detecting vulnerabilities based on the steps described extensively in Chapter three and whose results and validation are presented in subsequent chapters. The techniques adopted in this thesis are very rigorous and detailed given that they combines techniques such as static code analysis technique which has the capability to address problems like array bound check, uninitialized variables, unreachable code, syntax problems, undeclared variables, parameter type mismatch, uncalled function and procedure, non-usage of function results and misuse of pointers. Although both the signature technique used in anti-viruses and the techniques adopted in this work adopts database to match vulnerabilities, the current approach in this thesis examines the codes in order to address the vulnerability issues with a certain level of accuracy unlike in antivirus which only scan unknown files so as to get rid of the infected files.

2.9 Summary

This chapter has established the key concepts and theories which the current work explores, identifying the gap in knowledge that this research seeks to address. Specifically, the following summarises the key research aspects highlighted in this chapter:

(a) A brief overview of legacy systems where assembly language programming is still of invaluable and immense application.
Chapter 2: Background and Literature Review

(b) An overview of binary executable files in assembly language programming and their role in enhancing vulnerability threats in computer science and software engineering parlance. The justification for focusing on binary executable files was highlighted.

(c) A detailed review of software vulnerabilities in terms of types, detection tools, different sources and prevention techniques was provided.

(d) The step above allows for a review of research efforts by other researchers working on vulnerability detection in low-level programming languages to be explored. In doing so, the research gap which the current work seeks to address was highlighted.

(e) To close the gap in knowledge identified, a review of key concepts and techniques in software engineering was carried out, identifying the strengths and weaknesses of each technique which then informed the approach taken to realise the aim and objectives of the current work.

The implications of these concepts and theories as well as for the research questions devised for the current study are discussed in Chapter 3.
Chapter 3 Proposed Approach

Objectives

- Provide a general overview of the proposed framework.
- Describe a pilot study (i.e. an empirical study)
- Present research methodology.
- Discuss the contribution components.
- Propose vulnerability detection method as extended for WSL workbench.
- Describe the WSL workbench with extended components.
Chapter 3: Proposed Approach

This chapter presents an overview of the methodological framework and offers a detailed description of what each component does and how they connect with each other to realise the research objectives stated in Chapter 1. The chapter also elaborates on how a programming language with vulnerabilities detection is constructed and presents different approaches of how a vulnerabilities analyser can be used to identify vulnerable code. Before delving into the details of the chapter, it is important to provide an overview of the overall research design and structure upon which the current study is based.

3.1 Research Rationale

For any piece of research work, it is important to be able to offer an explanation of the importance of the study being carried out through the provision of arguments that are adjudged valid. In the context of the current work, the rationale is to address the problem of vulnerabilities in binary executable files of low-level programming language such as assembly language. Over the years, several techniques and models have been developed for the purpose of vulnerability detection in high-level programming languages. However, little research has been conducted towards the automatic detection of vulnerabilities in binary executable files in low-level programming language. Vulnerability detection and analysis seeks to ascertain whether a system contains potential flaws that could be exploited by any individual with a malicious intent. The focus is on binary executable files because it is usually the first point of attack by a potential hacker. The binary level of a system is the level through which loopholes can be easily exploited. A few researchers have carried out vulnerability detection insufficiently in low-level programming language as highlighted in Chapter 2. Most of these researchers employ informal methods to address the issue of vulnerability detection. Also, some commercial vendors have also produced tools for vulnerability detection in binary executable files but their methods and results remain unverified and unpublished. In the context of the current research, the overall focus is to adopt within an extended WSL framework through its transformation engine whilst using other software techniques such as program slicing combined with static taint analysis as well as single static assignment combined with value range analysis. The intended output will be an efficient and reliable automatic approach for vulnerability detection on binary executable files.

3.1.1 Pilot Study

In order to lay the foundation and gain an understanding of the overall research problem and establish a focus for the current work, there was a need to carry out a pilot study which stems from understanding key aspects of WSL and FermaT in terms of how they function and their
Chapter 3: Proposed Approach

capabilities and limitations. Findings from extensive literature search suggest that WSL can accomplish a wide array of tasks as compared to other programming tools especially as it pertains to its unique capability of supporting the migration from low level programming language and vice versa. It also supports the combination of both low and high level languages in terms of abstraction and construction. This is because WSL is endowed with a powerful and robust transformation engine which is based on formal methods that supports the optimisation of codes. WSL also has extensions which supports other tools and can provide greater details regarding the codes, whilst rendering assistance in terms of refinements. As such, WSL is a powerful tool as illustrated in Chapter 2 and Chapter 3.

After gaining an understanding of WSL in terms of its underlying structures and capabilities, a gap which stems from the limitations of WSL in terms of security during migration was identified and forms the basis of the decision to improve upon the analytic framework of WSL by incorporating new extensions that can identify and detect vulnerability pattern in low level language. However, as reported by a number of researchers, the incorporation of such extensions within WSL is a difficult proposition given the difficulty in tracing errors in low level language like assembly language. Against this backdrop, the idea to migrate assembly to WSL with the view to use FermaT to extract the vulnerability so as to help with refinements by using slicing techniques and SSA (Single Static Assignment), was conceived. This will enhance the detection of vulnerability issues in legacy systems that are powered by low level programming language such as assembly language, in industries. The idea to migrate assembly language to WSL directly was difficult to achieve, as such, the use of C programming language which was easier to understand was employed as translation technique which translate C language to WSL (C2WSL) in the first instance, and it was then modelled so as to yield similar behaviour with identical outputs. This was then followed by translation from assembly to WSL (ASM2WSL).

To ensure the veracity of ASM2WSL, we expanded upon the work developed by Pracner (2013) by upgrading it from 16 to 32 bits in order to meet the main requirements of translation and the idea was improved upon following the work of Ward (2005). The new model under development was tested using similar examples of C to disassembler with Intel in windows 7 to obtain MASM assembly code 8086 which is then linked to WSL. This then led to the identification of potential patterns of vulnerability by comparing the output from C and Assembly which gives an indication of the points where vulnerabilities are detected during migration. It was then established that in order to improve the accuracy of the vulnerability detection mechanism, it is important to identify the functions with the relevant codes that can detect vulnerability by knowing the size of the value when overwriting occurs. In order to
Chapter 3: Proposed Approach

actualise this, an extension tool for WSL which focus on a combination of taint analysis and FermaT slicing transformation method endowed with SSA and value range method for boundary checking was developed. Figure 3-1 shows the flowchart for the actualisation of initial mechanism of vulnerability detection ideas. Essentially, the pilot study was carried out to: gain an understanding of how vulnerability is attached to a file; observe the modifications made to a set of codes after infection takes place; develop mechanism for the extraction of vulnerabilities in the file; and to ascertain that all the requirements for vulnerability detection are met.

![Flow Chart for Vulnerability Detection within the Overall Model.](image)

As highlighted above, before performing the main data collocation procedure, a pilot study was carried out to test each component of the algorithm to ensure its functionalities work properly, and to establish the environmental requirements when collecting data, such as environment analysis (e.g. Memory Report, Vulnerability analysis Report), Call Graphs, metrics etc. Finally, the pilot results are then employed to derive and examine the vulnerability conditions, delivering results that are more accurate. Pilot data were collected while building the components. The vulnerability conditions were derived based on this data, and then the
Chapter 3: Proposed Approach

performance of the system was tested. The preliminary results indicated that validating the performance of the proposed algorithm required the collection of data that is sufficiently precise to test each component of the algorithm thoroughly. The data collection environment should vary with mix areas between memory analyser and vulnerability analyser. Thus, it is important to thoroughly test the algorithm and the decision maker system. The experiment also requires data to be collected using 60 different vulnerabilities, given the variance in the vulnerability forms. Full details and description of the operations of C language that were taken into consideration during the initial investigation is presented in Appendix D. Also presented in Appendix D is the classification of software vulnerabilities with requirement needed as well as system requirements and choice of programming language.

3.1.2 Vulnerability Detection Approach Adopted

One of the hallmarks of the current research is to enhance vulnerability detection in legacy systems whose underlying software platform is powered by assembly language programming. To achieve this, taint analysis a form of static analysis detection mechanism was integrated with transformation rules within WSL with the view to make detection of vulnerabilities more effective and efficient. MILLAR (2017) claimed that a number of researches have been carried out to realise this aim but most of them are not robust given that they are based on informal methods, thereby affecting the accuracy of their detection capabilities. As such in the current work, formal methods based on FermaT transformation engine within a WSL framework were adopted. This approach was used for two reasons namely:

(a) The implementation of FermaT within WSL is based on mathematical principles and formal methods. As such, the validity of vulnerability detection is guaranteed.

(b) In recent times, some programmers have tried to adopt transformation techniques based on program slicing for vulnerability detection and analysis based on informal methods (ZHANG, 2010). However, the approach resulted in a number of faults which further compounded the problem of vulnerability detection. The use of FermaT within WSL is therefore justified given that WSL is endowed with over 4000 transformation methods that are proven and well established. Its adoption in the current work therefore guarantees accurate results in vulnerability detection (Ward, 2013).
Chapter 3: Proposed Approach

In this research, the vulnerability detection focus mainly on two methods and sub-methods as indicated in Figure 3-2.

![Figure 3-2 Steps Involved in Vulnerability Detection](Key: VD- Vulnerability Detection; MA- Memory Analysis; VA- Vulnerability Analysis; TA- Taint Analysis; S-Slicing; VRA- Value Range Analysis; SSA- Static Single Assignment).

As shown in Figure 3-2, the vulnerability detection mechanism developed in this research is based on (i) Memory Analysis using, Value Range Analysis and Static Single Assignment and (ii) Vulnerability Detection based on program slicing combined with taint analysis. In the following section, a detailed description of the overall system framework is provided.

3.2 Research Design and Structure

The research design and structure which drives the various stages of the research process are depicted in Figure 3-3. This is more or less the ‘rules of engagement’ of the entire research work from the beginning to the end. The methodological strategies identified in the current work were implemented based on the specific research activities illustrated in Figure 3-3. The entire research is contextually driven, due to its cyclic nature which stems from the fact each activity that pertains to the research builds upon the understanding gained in previous activity(s). It provides feedback effects and loops that can assist in improving the preceding and future activity(s) further. As shown, the literature review is an integral and repetitive component of the research which assists towards the establishments of the knowledge base that is required to address the research problem.
In the section that follows, the overall approach taken to conduct the research is presented.

3.3 Research Approach

As highlighted earlier, the goal of this chapter is to describe the framework for vulnerability detection in binary executable files based on formal methods. Accordingly, the research approach taken is to investigate and establish a method for the identification of pattern of vulnerabilities in binary executable files of assembly language programming based on formal methods using Wide Spectrum Language (WSL) and its transformation engines. This is then followed by ascertaining competitive advantages that can be derived from developing tools that are based on WSL such as Fermat transformation engine and metrics as well as visualisation engine in comparison with what other researchers have done in this field. The overall focus is to come up with a robust and efficient method that will make vulnerabilities detection in binary...
Chapter 3: Proposed Approach

Executable files easier and reliable. It is well established WSL cannot directly detect vulnerabilities in low level language, as such it must be adapted as an intermediary language between high-level and low-level programming language (Ward, 2014). This is achieved through the superimposition of three programming languages including C, which is a high level programming language, WSL which is considered as an intermediate level language and a low level programming language (i.e. assembly language) the overall focus and research approach in the current work is summarised in Figure 3-4.

Figure 3-4: Overall Flow Chart Illustrating the Key Steps Involved in Vulnerability Detection Based on Contributions from the Current Work.
Chapter 3: Proposed Approach

As indicated in Figure 3-4, the first step is the development of a compiler which translates C codes into WSL (henceforth referred to as C2WSL) such that certain classes of vulnerability translate to certain semantics behaviours in the generated WSL code. Program transformations can then be used to isolate and analyse these semantic behaviours. Essentially, the translation from C to WSL which entails vulnerability analysis and translation allows for the easy identification and acknowledgement of the vulnerabilities pattern, given that high level language are easier to understand.

The second step involves the development of WSL vulnerabilities analysis framework with the view to identify vulnerabilities within the formal language. Specifically, in order to check for a possible buffer overflow on a string or array, for example, each assignment is translated to WSL code which checks the array boundaries and terminates the program with an error if the boundaries are exceeded. This program is semantically equivalent to the original, for initial states in which no buffer overflow occurs, but changes the semantics on buffer overflow to something which is easier to detect.

The third step is the development of a compiler to translate from Assembly to WSL (henceforth referred to as ASM2WSL) to further ascertain the pattern of the vulnerabilities. This compiler is constructed based on the translation results or outputs derived from other compiler from C2WSL. The outputs of C2WSL and ASM2WSL are compared to ascertain if there is consistency as to whether it meets the defined requirements for vulnerability detection in binary executable files. Given that the main gap which the current research work seeks to fill is vulnerability detection in binary executable files, it is important to establish the pattern of such vulnerabilities before a detection analysis framework can be constructed. Against this backdrop, the ASM2WSL framework in this work entails detection of vulnerabilities of the unknown exe binary file without the need to get the original source code. This is because the executable binary file will be decompiled into assembly which is later translated from assembly to WSL from where it can be ascertained whether vulnerabilities exists or not.

Furthermore, within the ASM2WSL framework, the outputs of C and Assembly in WSL are compared with some examples to ascertain and identify the pattern of the vulnerabilities. Common patterns based on WSL are then extracted and compared with their equivalents in C for consistency across all the three programming languages. Transformation method was employed within the overall framework in order to isolate the code for vulnerabilities detection without affecting the other nodes of the code as these transformations have been built based on formal method that relies on proven mathematical principles. Also the transformation method helps to minimise the number of lines of codes whilst aiding the restructuring of the translated
Chapter 3: Proposed Approach

code from assembly to WSL to make it look easy and simple to understand. Transformation technique such as program slicing to extract the related node of the vulnerability node was extensively employed. See Figure 3-5 for a pictorial representation of a framework for vulnerability detection based on buffer overflow.

![Figure 3-5: Pictorial Representation Translation for Vulnerability Detection Based](image)

To summarise, the overall framework for vulnerability detection in binary executable files in this research work entails the followings steps:

(a) Take existing code in one or more existing languages (e.g. C code).

(b) Translate this into WSL in such a way that certain classes of vulnerability translate to certain semantic behaviours in the generated WSL code.

(c) Use program transformations to isolate it.

(d) Analyse these semantic behaviours using dataflow analysis and metrics for easy detection of vulnerabilities.

3.4 Detailed System Framework

The development of a robust mechanism towards vulnerability detection in binary executable files in low-level programming language is non-trivial, given that such vulnerabilities are varied in scope, form, and function. Vulnerability detection is a complex aspect of the computing industry because of the level of havoc it can wreck on a given system. Despite these complexities, considerable amount of effort has been persistently geared towards the detection of vulnerabilities as precisely as possible. But given that computer systems such as legacy systems
Chapter 3: Proposed Approach

where assembly language codes are still being used undergo changes including alterations, extensions, and retrofits, due to their obsolete nature, it is difficult program them for minimal vulnerabilities with impeccable precision. The quest to find a way around these complexities prompts the embrace of approximate representations of observed vulnerabilities in systems such as legacy systems.

The first attempt at most approximations of any physical system is an illustrative description and understanding of the system being studied. Consequently, this helps in establishing the phenomena of logical assumptions necessary to limit the boundary of complexity of the system under consideration. With the assumptions in mind, inferences regarding the relationships between the system under observation and certain parameters as well as factors of interest, can be drawn. The goal of initial observations of the system and identification of appropriate assumptions is to provide the foundation and principles for the computational frameworks of the underlying phenomena being investigated. Consequently, these frameworks impose what can be described as a vast array of input-output relationships on certain variables in relation to others.

Figure 3-6: Overall System Framework for Vulnerability Detection.
Chapter 3: Proposed Approach

In the light of the above, the ensuing section and subsections gives an overall description of the overall framework adopted for the detection of vulnerability in legacy systems. The framework demonstrates the components of the proposed vulnerability detection system based on the concepts described in the preceding sections. Figure 3-6 gives a detailed representation of the systems components, including the way that these components are connected to each other in order to monitor the integrity of the system and provide a warning mechanism for the potential users of the overall system.

As indicated in Figure 3-6, the first block represents the source program which is where the initial code translation from high level language to low level language is carried out. In the current architecture, the transition from high level language to low level language is made possible through the use of a GNU Compiler Collection (GCC) as illustrated in Figure 3-7.

![Figure 3-7: An Illustrative Example how Source Code Transform between Different Programming Languages.](image)

GCC which is a form of compiler that has the facilities to support different programming languages as well as possessing the ability to generate assemble codes into AT&T syntax was adopted to disassemble the C execution binary file to assembly language. Its command can be used for translation purposes, compilation of executable or object code file and the disassembling of the object code in Intel ASM. The GCC can effectively produce the assembly file. In the overall framework of the current research, C language a high level language was adopted because it is relatively easy to understand and it has been adopted by many popular computer firms in the past for vulnerability detection analysis. Additionally, given that the C language has some inherent vulnerability issues such as buffer overflow which can be explored and understood, it was made the programming language of choice to assist in the current investigation. Accordingly, within the overall framework, vulnerabilities examples in C such as buffer overflow can be executed to obtain the binary executable file which can then be decompiled into the GNU compiler collection. Given that the detection of vulnerability in low
Chapter 3: Proposed Approach

level language such as assembly language is very difficult, the concept of re-engineering derived from the use of WSL for the purposes of translating low level language into a level that possesses both attributes and constructs of low-level and high-level languages was adopted within the overall framework. The inherent characteristics and advantages of WSL is already documented in Chapter Two. After the transitioning between low level and high level language based on the source codes are accomplished, the next task within the model entails the process of translation as described in the section that follows.

This is the next block in the overall framework as illustrated in Figure 3-6. In the framework, the overall plan is to decompile existing vulnerabilities code in C language by taking the binary executable file of the C code, to obtain the assembly code which is then modelled into WSL. This will be carried out in such a way that certain classes of vulnerability translate to certain semantic behaviours in the generated WSL code so that the vulnerabilities in the existing code can be determined through the analysis of the WSL. In the framework, buffer overflows were modelled in WSL with the view to define a single array or sequence which models the memory of the C program. In this framework, a C variable, string or array is represented by a subsequence in the memory model. The overall purpose of translation is to allow for easy modelling of vulnerabilities into WSL, enable the translation of C and the assembly x86 to WSL and to allow for the extension of the existing translator to translate assembly x86 self-contained 32 bits to WSL. This procedure also allows for the checking of vulnerability during the translation activities. The translation of exe File, C File, ASM File to WSL allows for the modelling of the original code into WSL so that the functions and the lines of behaviour causing the vulnerabilities can be ascertained. The techniques used for translation is based on static analysis including: taint analysis; tagging and marking of the relevant functions/components of the vulnerabilities and the modelling of vulnerabilities behaviour which can assist in making clarifications to various forms of vulnerabilities.

This FermaT Maintenance Environment is the component of the overall framework where the actual program transformation is taking place based on the highly defined mathematical representations in WSL and FermaT. The graphic user interface of FermaT Program Transformation System is called the primary FermaT Maintenance Environment. Consisting of a text editor, it has the capability to express WSL within an Abstract Syntax Tree (AST) viewer. Through the AST, a programmer can navigate through the source codes using a console provided by the environment of FermaT transformation engine. Such transformations can generate semantics that are equivalent or a refined version of the constructs of the source program. A communication pipe which is part of the underlying operating system upon which
Chapter 3: Proposed Approach

the model is working establishes communication between the FermaT transformation engine and the FermaT maintenance environment. Full details of how these are achieved within the overall framework are provided in Appendix B.

Detecting vulnerabilities in WSL code currently does not have many useful applications. As such, in the current work, a vulnerability detection tool for WSL was constructed which help in detecting the potential vulnerabilities so that functions, assignments or variables can be traced using FermaT transformation rules. In the model, a classification scheme based on two dimensions namely vulnerabilities and potential perpetrators and identification of the features or conditions of the vulnerabilities are presented. The use of the security engine is for detecting and analysing the vulnerability in term of the program within its boundary and makes the decision of both outputs. The algorithm is built to detect the buffer overflow and check the location in the memory where the data saving on registration and address.

3.5 Summary

In this chapter, the overall research design and methodological approach as well as system architecture for vulnerability detection in binary executable files based on extended WSL and FermaT transformation technique was presented. The system architecture is composed of three subsystems namely security engine, application subsystem and database which were developed based on the principles of vulnerability detection in low-level programming language with a focus on binary executable files. In addition, the components of the integrity process were designed using the abstract layered vulnerability detection framework. The vulnerability detection algorithm was integrated within the security engine layer in order to ascertain the level of vulnerability of the acquired data. The main purpose of the proposed system is to provide a robust and reliable mechanism for checking the vulnerabilities in binary executable files and detecting any misleading information based on formal methods and analysis. Developing a vulnerability detection framework requires a great deal of procedures and principles. As such, in Chapter Four, a detailed description of the development of each component based on vulnerabilities detection algorithm is presented.
Chapter 4 Process of Vulnerability Detection Approach

Objectives

- Propose a novel vulnerability detection approach for systems powered by low-level programming language.
- Describe the architecture of the vulnerability detection mechanism.
- Describe the components of the multi phases of the proposed algorithm.
- Describe the subsystems of the architecture and its components.
- Provide the processes that are used to detect program vulnerabilities.
- Describe the adaptive algorithm upon which the WSL is based to detect program vulnerabilities.
Chapter 4: Process of Vulnerability Detection Approach

This chapter presents a detailed description of the algorithms and structure of the framework adopted for vulnerability detection in binary executable files powered by assembly language programming. The chapter describes in detail each component in terms of how they were adopted to fulfil the overall requirements of this research. Next, it provides an explanation to every component of the framework, the underlying rationale and applicability for vulnerability detection system. The chapter concludes with a list of vulnerability conditions representing buffer overflow behaviour and how they can be detected using vulnerability detection mechanism based on WSL within a FermaT engine.

4.1 Vulnerability Detection Process

This section presents the steps involved in the development of the vulnerability detection algorithm. The algorithm has the ability to perform three phases of integrity checks. A detailed architecture for the vulnerability detection mechanism is depicted in Figure 4-1.

Figure 4-1: Architecture Illustrating the Vulnerability Detection Process.
Chapter 4: Process of Vulnerability Detection Approach

Vulnerability detection algorithm has been developed and integrated into the translation process and FermaT. As shown in Figure 4-1, the first step in the process is to take an input source program file and translate it into WSL action system. As highlighted in Chapter Three, the focus is on two programming languages namely C programming language which is a high level language that has some inherent vulnerabilities and assembly language which serves the programming language upon which the legacy systems work. Therefore, the level of the code file is checked first and prepared for translation processes. The translation process is developed based on taint analysis to help in tracing, at a first instance, the potential vulnerability in a system based on syntax analysis and correlation of semantics in order to establish communication between various components of the system. This can then allow untrusted functions in the database in terms of logs and security policy to be classified, thereby helping to focus on vulnerable functions for the next level of vulnerability detection. The detail of this aspect of the model framework is presented in the succeeding sections.

4.2 Translation Process Phase

The first phase in the proposed translation process algorithm is to translate the input source program to WSL, in order to detect the vulnerability within the binary executable file. To ensure the current translation method is correct, the existing vulnerabilities in C language need to be compiled to generate the binary executable files which are then decompiled to obtain the assembly code based on the command shown below.

```
Compile to .exe file (The following command generate “exe” file)
*****************************
$ gcc -o filename filename.c
*****************************
Then Compile to Assembly file
*****************************
$ gcc -S filename.c
or
$ gcc -S -masm=intel filename.c
```

Listing 4-1 GCC Decompile Commands

The overall purpose is to translate the output code into WSL in such a way that certain classes of vulnerability are translated into certain semantic behaviours in the generated WSL code, so that the vulnerabilities in the existing code can be determined by the analysis of the WSL.

High/low-level modelling of the program is required by focusing on the memory leak vulnerability called buffer overflow. In order to model buffer overflows in WSL, a single array or sequence need to be defined which models the memory of the C program. In the current
model, a C variable, string or array is represented by a sequence in the memory model. Process of migration is based on two factors namely: classification of software vulnerabilities and translation and modelling of the behaviour of vulnerability. Figure 4-2 shows the process of the translation in order to actualise the two factors highlighted with brief description of the underlying components provided in the section that follows.

**Figure 4-2: Process of Translation.**

(a) **Lexical Analysis**

Lexical analysis entails the breaking down of source codes into tokens (i.e. smaller components of the primary code) with each token representing an entity of the language such as an identifier. The syntax of a token is in the form of a normal language such that it can be easily recognisable within a set of regular expressions. The software responsible for such an analysis is termed a lexical analyser or a scanner. The art of lexing assist in getting rid of whitespaces and helps in transforming the source into a stream of tokens that are processed by a parser.

(b) **Syntax Analysis**

Syntax analysis involves parsing the token sequence for the identification of the basic syntax of the overall structure of the program. This phase typically constructs a parse tree, which
substitutes the linear sequence of tokens with a tree structure constructed based on the set of rules which forms the basis of the syntax of the programming language. The analysis of the syntax of the source code is carried out by the parser whilst ensuring that it conforms to the language grammar. Errors based on syntax are detected and reported during this phase of the compiler. The parser’s output is an abstract syntax tree (AST) representation of the program.

The abstract syntax tree representation visibly provides a definition regarding the syntax of the program and is adopted for a number of functions including type checking, semantic analysis, and high-level optimisations. The tree consists of tree nodes, designed to express the structure of the language. An example of an AST is given in Figure 4-3.

![Abstract Syntax Tree](image)

**Figure 4-3:** Abstract Syntax (AST) Tree Representation of \( C = A + B \).

(c) Semantic Analysis

In this phase, semantic information are added by the compiler to the parse tree, leading to the construction of the symbol table. Semantic checking routines including object binding, type checking, incorrect program rejection, issue of warnings and lots more are carried out in this phase. A complete parse tree is required for semantic analysis and it logically follows the parsing phase, but precedes the code generation phase.

(d) Code Optimisation

Here, optimisation of the code is carried out by the intermediate code and it entails a routine whereby line of codes that are deemed unnecessary are purged whilst organising the sequence of statements in an orderly fashion with the view to enhance the execution of programs and minimise the use of computer resources such as processing speed and memory utilisation.

(e) Code Generation
Chapter 4: Process of Vulnerability Detection Approach

Here, the representation of the intermediate code that are already optimised are taken and mapped to the machine language that is targeted. The code generator performs the translation of the intermediate code into a sequence of machine code that can be relocated or reassigned.

(f) Taint Analysis

This is based on the same principles described in Chapter three and is schematically illustrated as shown in Figure 4-4.

Taint analysis is employed in the current work to ascertain the locations of programs in instances where data is read from sources that are untrusted and where data that are tainted are used as an argument to a function that is vulnerable. In this framework, two lattice values namely NOT TAINTED and TAINTED are used. The meet function for the lattice is based on the criterion that anything that meets a tainted value is flagged as tainted.

\[
\text{NOT VULNERABLE} \land \text{NOT VULNERABLE} = \text{NOT VULNERABLE} \\
\text{ANYTHING} \land \text{VULNERABLE} = \text{VULNERABLE}
\]
Chapter 4: Process of Vulnerability Detection Approach

The description of the adaptive process is given below:

1. Initialise all variables as NOT VULNERABLE.
2. Find suspicious constructs and specification (function and expression). Mark the values returned by the function as VULNERABLE.
3. Propagate the tainted values through the program. If a tainted value is used in an expression, mark the result of the expression as VULNERABLE.
4. Repeat step 2 until end code point is reached.

Listing 4-2 Model the assembly instructions and memory to WSL.

For details of the characteristics displayed by the patterns under investigation, see Appendix C.

4.3 FermaT Phase

The second phase in the proposed method is to check the output of the translation process and determine precisely the transformation process that will be required to restructure the code from WSL Action System to basic WSL as depicted in Figure 4-5. For detailed explanation of the building blocks of Figure 4-5 above, see Appendix C.

![Figure 4-5: The Transformation Process for Restructuring.](image)

4.4 Vulnerability Detection Phase

Vulnerability detection was adopted as a static technique which is applied to the source code when the application is not being executed, so that specific information can be derived from the source code. The source code is checked based on two stages: (i) through Vulnerability Analyser, which looks for the functions and expression and statements that might cause the vulnerability, after which it is passed to the next stage called (ii) Memory Analyser which looks for the arrays
Chapter 4: Process of Vulnerability Detection Approach

and the input value and check if there is any effect on the memory in order to ensure that the code work properly with no memory leak. With both result from the two stages, the level of the risk that might cause potential vulnerability can be ascertained. Figure 4-6 gives a pictorial representation of the steps involved in the vulnerability detection phase. For prototype design of the building architecture of Figure 4-6 below, see Appendix D.

![Figure 4-6: Process for Vulnerability Detection.](image)

In the subsections that follow, a brief explanation of each of the building blocks in Figure 4-6 is provided.

(a) WSL Vulnerability Analyser Technique (WVAT)

As highlighted earlier, detecting vulnerabilities in WSL code currently does not have many useful applications. As such a taint analysis tool was built for WSL to enhance its vulnerability detection capabilities so that attributes such as functions, assignments or variables can be traced by using FermaT transformation rule. In this work, WSL Vulnerabilities Checker combining taint analysis with slicing was used to analyse the WSL code based on the relation:

\[
\text{Risk} = \text{Likelihood of Threats} \times \text{Potential Vulnerability} \times \text{Type of Impact}
\]
Chapter 4: Process of Vulnerability Detection Approach

Type of Impact = Likelihood of Threats × Potential Vulnerability

**Taint analysis**: Identify the Potential Vulnerability.

**Slicing**: Simplify the identified threats from source code.

The overall schematic framework for the vulnerability analyser is depicted in Figure 4-7.

![Diagram of vulnerability analyser process]

**Figure 4-7: Components of Vulnerability Analyser**

The process for vulnerability analyser is illustrated in Listing 4-3.

**Input**: WSL source code

**The adaptive Process of vulnerability analyser**

1. Apply Lexical Analyser (Scanner) on WSL code (converting a sequence of characters into a sequence of tokens by removing any whitespace or comments in the source code)
2. Apply Syntax/Semantic Analyser (Parser) on output from 1
3. Using taint analysis initialize all variables as Safe
4. Find functions or expression that may be an untrusted.
5. Check functions or expression. If a tainted value is used, mark the result of the functions or expression as Vulnerable.
6. Slicing for the tainted functions or expression
7. Repeat step 5 until the end of the code.
8. Find all vulnerable functions and report it as vulnerability.
9. Report Alarm-Unit as an exist potential vulnerability.
Chapter 4: Process of Vulnerability Detection Approach

Listing 4-3 Model the assembly instructions and memory to WSL.

**Output:** WSL Code with vulnerability labelling

- **Vulnerability Conditions**

  Vulnerability conditions have been identified that will be used in order to detect the vulnerabilities and if these conditions are met, then the code is adjudged safe and secure otherwise they are considered as vulnerable.

  **Table 4-1 VCs Useful for Detecting Vulnerable Code.**

<table>
<thead>
<tr>
<th>Description</th>
<th>Vulnerability Condition(VC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If variable size of source code less or equal the destination reserved for variable size.</td>
<td>Src_VarSize ≤ Dst_VarMaxSize</td>
</tr>
<tr>
<td>If copies up to n characters from the source string pointed by source to destination less or equal the destination reserved for variable size.</td>
<td>Min(Src_VarSize, n.length) ≤ Dst_VarSize</td>
</tr>
<tr>
<td>If the input known and size of it less or equal the destination reserved for variable size. In case of copy string in variable.</td>
<td>Input = known, Src_Size(&quot;Input&quot;) ≤ Des_MaxSize</td>
</tr>
<tr>
<td>If the buffer destination reserved size index larger or equal to variable string size of the source. appends the variable string pointed to by source to the end of the string pointed to by destination.</td>
<td>Buff_Size + Var_Str_Size ≤ Buff_MaxSize - 1</td>
</tr>
<tr>
<td>If the array size larger than the variable size.</td>
<td>Block_Array_Size &gt; VarSize</td>
</tr>
<tr>
<td>If variable size X of destination known reserved size larger or equal the unknown infinite variable size.</td>
<td>(\infty \leq x_Size)</td>
</tr>
<tr>
<td>If print source variable size less or equal the destination reserved for variable size.</td>
<td>Prnt_Src_VarSize ≤ prnt_Dest_Size</td>
</tr>
<tr>
<td>If size of first source code equal to second source code then copy n characters from memory area second source code string to memory area first source code string.</td>
<td>Var_Src1_Size == Var_Src2_Size, Index_Var_Src1 = (Var_Src2.n_length)</td>
</tr>
</tbody>
</table>
Chapter 4: Process of Vulnerability Detection Approach

- Slicing via FermaT Transformations: Value Flow

In order to gain an understanding of a given program, it is required to ascertain how variables are assigned their values at specific points. This can be achieved by applying slicing method to extract/isolate the potential vulnerability code with the view to allow the developer to see the vulnerabilities clearly with its relations code and then different techniques such as on dataflow diagram or graphic call diagram can be adopted for getting more details. After this step, the extract code can be manipulated with the FermaT transformations which help to optimize and transform the extraction code from one form to other structure to see the vulnerabilities code from a different perspective. For example, slice of the program from the buffer assignment will automatically eliminate code which is not relevant to this particular assignment and reduce the amount of work needed to either prove the non-existence/existence of buffer overflow, and to compute an example input state and control flow path which may result in buffer overflow. The codes extracted with potential vulnerability are then sent to the “Memory Analyser” in order to analyse the boundary. Further theory and explanation on program slicing, forward and backward slicing, backwards syntactic and backwards semantic slicing are also provided in Appendix C.

(b) WSL Memory Analyser Technique (WMAT)

Applying the WVAT (i.e. combining taint analysis with slicing technique) to a program that is vulnerable assist in discovering the execution paths of an untrusted source from which data is read and used with malicious, representing two vulnerable characteristics to be detected. The third attributes of vulnerability arise when the data is not validated at a sufficient level. To address this in this work, boundary checking mechanism was built and it involve value size range that analyses the relevant component of vulnerabilities and dividing the variables or input separately so that each single segment in each index is clarified if there is any potential memory leak on the code by checking if statement confirms that the value of $n < \text{the size of the destination buffer}$, making the memory copy call safe.
Chapter 4: Process of Vulnerability Detection Approach

As indicated above, an analysis is required such that it takes into consideration the influence of an IF statement whilst determining the possible values of the n variable within the THEN and ELSE loop. This was achieved through the calculation of a range of possible values for individual variables. The algorithm operates based on Static Single Assignment (SSA) by representing each definition of a variable as a distinct version, giving the algorithm the ability to differentiate between the various ranges of n within the two branches of the IF statement.

- **SSA via FermaT Transformations: Variable Version**

  Modern day compilers are endowed with the capability to transform a program into an SSA form before program optimisation takes place. The SSA forms possess the attributes that each variable in the program is the target of only one assignment. Transformation is carried out through the creation of various versions of a variable, each representing an assignment. The SSA form of a program gives room for optimisation to be carried out in an efficient manner, simplifying the implementation procedure in the process. Given that unrelated variables are assigned new variable names the use of data flow analysis to establish their relationship will no longer be necessary. The benefits of SSA for the purpose of program optimisation have been extensively demonstrated in extant literature. SSA exemplifies a program in which each variable

![Diagram](image-url)
Chapter 4: Process of Vulnerability Detection Approach

is assigned only once. To convert to SSA form we rename the variables and add "phi functions". SSA provides a concise representation of the data flows in a program given a variable reference when it is required to find all the assignments to the variable which could reach this reference. This is a relatively easier task that can be accomplished using SSA form given that there is at most one assignment to the variable. Phi functions are required when two or more assignments to a variable can reach a reference. For example:

\[
\text{IF } y = 0 \\
\quad \text{THEN } x := 1 \\
\quad \text{ELSE } x := 2 \text{ FI;}
\]

\[
z := x
\]

Listing 4-4 Fragment of WSL Code

In the statement \( z := x \) the value of \( x \) may have been assigned in the first or second branch of the IF. A possible SSA form is:

\[
\text{IF } y_0 = 0 \\
\quad \text{THEN } x_1 := 1 \\
\quad \text{ELSE } x_2 := 2 \text{ FI;}
\]

\[
x_3 := \phi(x_1, x_2); \\
z_1 := x_3
\]

Listing 4-5 Applied the SSA Transformation to the Code

There is now only one assignment to each of the (renamed) variables. The variable \( x_3 \) and the statement:

\[
x_3 := \phi(x_1, x_2);
\]

are added to handle the control flow join point after the IF condition. Control flow cannot be predicted in advance, so it is not always possible to know which definition of a variable reached is applicable for a particular use. To handle this uncertainty, phi functions are created and are placed at the control-flow joins. In this case, the value of \( x \) may proceed from the argument or from the assignment in the first or second IF statement and it is easily indicated by the phi function.

To ascertain if the use of \( n \) is vulnerable, the upper bound of \( n \) at that location has to be compared to the destination buffer size. If a buffer is allocated dynamically at runtime and its size is not known during the static analysis, it will be assumed if the upper bound equals the maximum integer, then there exist vulnerability. Also, the data location in the memory where the data is saved must be checked for vulnerability. Analysis checks the index and boundary of the
array for a possible buffer overflow on a string or array. See below for steps illustrating the procedure.

**Input:** WSL Source Code

The adaptive process of Memory Analysers to determine value size range for each variable.

1. Find function or expression that read data from an untrusted source that access the heap.
2. Consider the size of the variable as threshold.
3. Check if the variables have been overwritten.
4. Place the new value of the variable and compare it with the threshold.
5. Check if the function access locations saving in the dump heap.
6. If an argument is VULNERABLE and its range is outside the safe range for the function.
7. Report vulnerable functions as vulnerability in WSL.

Listing 4-6 Model the assembly instructions and memory to WSL.

**Output:** WSL Code with boundary vulnerability labelling

```c
char *A = (char*)malloc(n1*sizeof(int));
```

Listing 4-8 Heap Command

1. ∀ f, where f functions or expression at least 2 variable x and y
2. If f copies y to x, then check
   a. y ≤ len where len is length of the variable, then it is safe
   b. x < len < y, then possible vulnerability exist
   c. len < y, then vulnerability exist
   d. if x[-1] or x[+1] value change then vulnerability exist

Boundary checking = (variable index - 1) + (variable index + 1)

Listing 4-7 Formalisation of Memory Analysis

If the new size is greater than the threshold, then it considered as vulnerability exist. According to the rules of Listing 4-8, buffer overflow would not be considered to happen absolutely in case y ≤ len implying that estimated min value of the activity area is less than allocated length, which is impossible in case len < y meaning that max value of activity area is greater than allocated length, otherwise possible. The following command has been employed to reserve a place in the heap for checking the boundary of the value based on the algorithm described above. The n1 consider the size of the destination variable for example copy (x, y); x is destination variable. To ascertain that a buffer overflow has occurred, the index in an array must be seen to behave as a memory location after which values are added to check if the code is saved outside the defined boundary or not. If this behaviour occurs, then it can be established that the assembly code which has been translated into WSL has vulnerability. In this research work, memory analysis is developed to check the code that has been detected from the
vulnerability analyser and whether it will affect the memory or not, based on boundary checking method for the WSL. WSL is translated to Scheme for interpreting or compiling and Scheme is based on static binding. Dynamic binding is emulated in Scheme as follows: On entry to a VAR block, the global value of a local variable is stored in a Scheme static bound local variable. Then the global variable is assigned the new value. On exit of the VAR block, the global variable's value is restored from the local variable.

Given that a WSL string is allocated on the heap and can be of any length, a memory analysis was built to check the boundary of the detected variables. There is no way to directly access the stack or heap from WSL. As a result, the WMAT technique was developed to enhance the performance of checking the boundary at a quicker pace and with ease in order to trace the memory size of buffer-related variables.

Heap memory is usually divided and allocated in chunks which contain memory management information within them. When a heap buffer overflow occurs, it attempts to overwrite the memory management information of the next chunk in memory with specific values in order to transfer control to the attack code. Overflowing the heap are either by overwriting function pointers or virtual function pointers. The increment values will be considered as dump. Figure 4-9 depicts a procedure illustrating how the proposed method for the memory analyser works.
Chapter 4: Process of Vulnerability Detection Approach

Figure 4-9: Illustration of the Mode of Operation of the Proposed Method for Memory Analyser

The procedure on how to calculate the effective size of a Stack Memory is described in Figure 4-10 below. Let SM be the Stack Memory and V be a variable or the Input. Then:

\[ SM = V \setminus \{\text{Size of Array}\} \]

Figure 4-10: Calculation Procedure of the Effective Size of a Stack Memory.

The result of the memory analyser will migrate to the alert unit to help the decision maker to issue the report for the final stage.

(c) Decision Maker Phase

A combination of the results from both vulnerability analyser and memory analyser can provide a guaranteed and secured assessment of the vulnerability code. Figure 4-11 shows the diagrammatic illustration of the decision maker phase integrating both memory and vulnerability analyser.
Chapter 4: Process of Vulnerability Detection Approach

![Diagram of Decision Maker Phase]

Figure 4-11: Framework for Decision Maker Phase

The vulnerability analyser and memory analyser results are combined and compared with each other. The results are taken from the alert-unit component and provide warning as to whether the program report vulnerability or non-vulnerability. The input factors are: (i) the vulnerability in vulnerability analyser ($VA$), (ii) the vulnerability in memory analyser ($MA$), and for $VA$ and $MA$ are non_exist, possible_exist and exist. The output of the decision maker is the vulnerability detection, which refers to the confidence level of the detection process. A detailed description of the rules is presented below.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>$VA$ is non-exist and $MA$ is non-exist then potential of vulnerability is non-exist</td>
</tr>
<tr>
<td>R2</td>
<td>$VA$ is non-exist and $MA$ is exist then potential of vulnerability is non-exist</td>
</tr>
<tr>
<td>R3</td>
<td>$VA$ is exist and $MA$ is non-exist then potential of vulnerability is possible_exist</td>
</tr>
<tr>
<td>R4</td>
<td>$VA$ is exist and $MA$ is exist then potential of vulnerability is exist</td>
</tr>
<tr>
<td>R5</td>
<td>$VA$ is not checked “possible_exist” and $MA$ is not checked “possible_exist” then potential of vulnerability is possible_exist</td>
</tr>
<tr>
<td>R6</td>
<td>$VA$ is not checked “possible_exist” and $MA$ is exist then potential of vulnerability is exist</td>
</tr>
<tr>
<td>R7</td>
<td>$VA$ is not checked “possible_exist” and $MA$ is non-exist then potential of vulnerability is non-exist</td>
</tr>
<tr>
<td>R8</td>
<td>$VA$ is exist and $MA$ is not checked “possible_exist” then potential of vulnerability is exist</td>
</tr>
<tr>
<td>R9</td>
<td>$VA$ is non-exist and $MA$ is not checked “possible_exist” then potential of vulnerability is non-exist</td>
</tr>
</tbody>
</table>

Listing 4-9 Model the assembly instructions and memory to WSL.
Chapter 4: Process of Vulnerability Detection Approach

Table 4-2 Vulnerability Analyser x Memory Analyser.

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>Decision Maker Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability Analyser</td>
<td>Memory Analyser</td>
<td>Decision Maker Result</td>
</tr>
<tr>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>Exist</td>
<td>Non-Exist</td>
<td>Possible_Exist</td>
</tr>
<tr>
<td>Exist</td>
<td>Not checked “Possible_Exist”</td>
<td>Exist</td>
</tr>
<tr>
<td>Non-Exist</td>
<td>Exist</td>
<td>Non-Exist</td>
</tr>
<tr>
<td>Non-Exist</td>
<td>Not checked “Possible_Exist”</td>
<td>Non-Exist</td>
</tr>
<tr>
<td>Non-Exist</td>
<td>Non-Exist</td>
<td>Non-Exist</td>
</tr>
<tr>
<td>Not checked “Possible_Exist”</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>Not checked “Possible_Exist”</td>
<td>Not checked “Possible_Exist”</td>
<td>Possible_Exist</td>
</tr>
<tr>
<td>Not checked “Possible_Exist”</td>
<td>Non-Exist</td>
<td>Non-Exist</td>
</tr>
</tbody>
</table>

The decision maker which is a logic-based BOF vulnerability auditor is used so that the quality of the warning signal can be improved. Against this backdrop, one of the hallmarks and contribution to knowledge of the current work is the development of BOF vulnerability analysers and the corresponding decision maker sets which serve as a viable tool for vulnerability detection in low level language. Additionally, for the overall assessment of the level of vulnerability of a program, a logic-driven multi-unit decision-making system was designed and implemented.

4.5 Summary

In this chapter, the steps involved in the development of vulnerability detection algorithm for binary executable files in low-level programming language such as assembly language programming is presented. The overall aim was to ensure that the navigation of the vulnerability detection program is consistent and alerts are triggered to notify a potential program maintainer when the program is not trustworthy. The chapter described the ways in which the proposed algorithm is able to reason out the integrity of the system by considering the vulnerability associated with positioning data, read data and the copy data. The chapter highlights the fact that improvement and enhancement/extension of the translation techniques is needed, and the current work contributes by extending the FermaT tools for WSL towards vulnerability detection in binary executable files of low-level programming languages. Boundary checking was built and extended to WSL to detect the buffer overflow. The chapter also illustrates how WSL and FermaT transformation Engine were integrated with vulnerability analyzer to define the patterns of vulnerabilities. Three phases of vulnerability detection were developed in the algorithm. This include WSL vulnerability analyzer technique (WVAT) algorithm, which was used to find the
Chapter 4: Process of Vulnerability Detection Approach

vulnerability functions; WSL memory analyzer technique (WMAT) which was used to check the memory leak as to whether it occurs or not; the decision maker (DM) which improves the performance of the system while ascertaining that vulnerability exist or not. This is important because in the WSL, it is important to ensure that vulnerability exists or not given that it is an intermediate language that interacts with low level language which requires additional level of detail to attain some level of accuracy. In the chapter that follows, different vulnerability will be examined, from which evaluation of results will be discussed.
Chapter 5 Vulnerability Patterns and Detection

Objectives

- Develop a prototype implementation of vulnerabilities detection scheme.
- Provide the samples used in this research to analyze the vulnerability and non-vulnerability through detection processes.
- Provide the functions that are used to detect program vulnerabilities.
Chapter 5: Vulnerability Patterns and Detection

This chapter begins through a pilot study which explains how FermaT transformation engine could be utilised to detect the vulnerabilities by means of their vulnerable functions. It provides a detailed explanation regarding which tools should be used to extract the vulnerability functions that represent the relevant vulnerabilities behaviour at formal method level. For instance, WSL with FermaT examines both vulnerability functions and vulnerability paths in two phases namely vulnerability functions scanning and bound checking of static analysis. In the context of the current work, the main type of vulnerability which is targeted is memory loss due to buffer overflow. This is important given that buffer overflow is the first point of call for most vulnerabilities attack. As highlighted in Chapter Two, two stages were identified where vulnerabilities are caused due to memory leak. First, the attacker finds a programming language or compiler that carries out non-bounds checking such as C language and assembly to unleash a malicious attack on computer systems. Second, the attacker obtains the information of the wanted unsecure functions a vulnerable to buffer overflow which use OS-level non-preventative functionality.

In this chapter, a formal framework for vulnerability detection as described in Chapter Three is presented. The framework demonstrate an automatic technique based upon both bounded model checking and taint patterns analysis with slicing within a WSL environment to discover memory leak vulnerabilities. If vulnerability exists, the technique presented in this work produces a trace of function operations demonstrating an attack by using the FermaT slicing transformation formal method and its efficacy is demonstrated using case studies presented in Chapter Six. Also adopted is a combination of single static assignment with value range analysis. Overall, the current research provides a novel way to analyse common family of buffer overflow format attacks, and an implementation strategy for the automatic detection of vulnerability. Against this backdrop, this chapter explains the implementation of the prototype model both at Translation and Vulnerability Detection level, whilst enhancing the attributes of WSL for effective vulnerability detection using FermaT techniques.

The motivation of the current work therefore stems from a desire to improve upon current static analysis techniques for detection of vulnerability with a focus on low level language. The work leverages on techniques derived from software vulnerabilities challenges and FermaT technology and has proven to be quite beneficial as highlighted in the succeeding section of this Chapter. The overall research started by analysing samples of programs with vulnerabilities, identifying similarities in the source code of the programs contaminated with vulnerabilities, which then led to the identification of a many patterns of source code responsible for the majority of the observed vulnerabilities. In this Chapter a vulnerability classification based on
Chapter 5: Vulnerability Patterns and Detection

these results with requirement needed is presented. In the analysis presented, it was discovered that many of the vulnerabilities the samples of code which were investigated were due to program execution path based on three common features which are highlighted in the succeeding section. The design and implementation of the technique employed in the design and implementation are presented in sections 5.2.

5.1 Vulnerability Features

After we compare 42 examples we identified six factors or features that can help to build our research detection method.

Table 5-1 Vulnerability Features

<table>
<thead>
<tr>
<th>Description</th>
<th>Fundamentals Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call functions: Track and check the function names</td>
<td>Call _Function_Name</td>
</tr>
<tr>
<td>Size of the full memory: make space for local variables</td>
<td>Sub esp, Memory_Size</td>
</tr>
<tr>
<td>Size of the used memory: size of the minimum address number that been insert the hex number sub the maximum address number that stored 0. Also we can calculate it from the load effective address with the minimum address number sub the maximum address that stored 0</td>
<td>“Mov Size.Input PTR [esp+Min(No.ofAddress)], Hex_No.” - “Mov Size.Input PTR [esp+Max(No.ofAddress)],0”</td>
</tr>
<tr>
<td>Size of the free memory: just after the address of the maximum address number that stored 0 until the Size of the full memory</td>
<td>[esp+Max(No.ofAddress+1)],0 until the esp, Memory_Size</td>
</tr>
</tbody>
</table>
| Values stored in the memory: normally the values stored in the beginning address of the extended stack pointer esp | mov DWORD PTR [esp+4], Values          
               mov DWORD PTR [esp+8], Values          
               and upper                             |
| Outsource input: the unknown data from user. | mov eax, DWORD PTR __imp___iob                                                     |

The justification for setting the following rules is to detect the vulnerabilities that could cause the system problems. The rules based on analysed code and data in order to understand the vulnerabilities how it has been hacked.

Table 5-2 Assembly Vulnerability Rules

<table>
<thead>
<tr>
<th>Description</th>
<th>Assembly Vulnerability Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the function name equal the vulnerability function name.</td>
<td>Func_Name == Vul_Func</td>
</tr>
<tr>
<td>If the extended stack pointer for the memory size less than space for the local variables.</td>
<td>Esp, (Ms) &lt; [Esp+Max(No.ofAddress)], values</td>
</tr>
<tr>
<td>If the reserved variable size less than the length of the input variable.</td>
<td>Reserved.Vs &lt; Veriable.Length([esp+Max(No.ofAddress)],0) - (lea eax, [esp+Min(No.ofAddress)])</td>
</tr>
<tr>
<td>If the Input variable size greater than the length of the input variable.</td>
<td>Input.Vs &gt; Veriable.Length([esp+Max(No.ofAddress)],0) - (lea eax, [esp+Min(No.ofAddress)])</td>
</tr>
<tr>
<td>If the free local space that will be used not equal to the reserved value space</td>
<td>([Esp, (Ms)] - ([Esp+Max(No.ofAddress+1)],0) ≠ [Esp+Min(No.ofAddress)], values)</td>
</tr>
<tr>
<td>If the beginning address values for the extended stack pointer not empty</td>
<td>[esp+Min(No.ofAddress)] ≠ Empty OR</td>
</tr>
<tr>
<td>If input command used</td>
<td>imp___iob == Exist</td>
</tr>
</tbody>
</table>
Chapter 5: Vulnerability Patterns and Detection

These rules can help to analyse the assembly code while modelling the suspicious code behaviours to WSL in order to understand the code easily for further analysis. The benefits of WSL is that can help to understand the language easily, it can help also for analysing the migrating code as the language supported with robots maintenance tools such as transformation. The transformation helps us to simplify the code with refinement. However, we have used also couples of transformations with taint analysis to help detecting the vulnerability accurately.

5.2 Proposed Solution

In this research, techniques regarding how to identify potential vulnerability in binary executable files bin assembly language programming was demonstrated by comparing assembly x86 containing vulnerability with those without vulnerability. This was achieved by comparing vulnerability functions and non-vulnerability functions in C programming language in order to identify the differences between each function call. The instruction sets in assembly language were also compared for vulnerabilities with the view to get rid of them. Most vulnerability in C is related to buffer overflows and string manipulation. In most cases, this would result in segmentation given that C programs were adapted as an example to confirm that the disassembled generated code having buffer overflow. This is then followed by looking into the assembly code that is generated from the disassembler. To detect the potential vulnerability buffer overflow in assembly, Functions identifier such as call function name (call _gets, call _strcy...so on) is required. And Libc functions identifier for instance to look for the code contents in program instructions. The overall principle is illustrated in the example shown in Table 5-3:

Table 5-3 Illustrate Example with and without Vulnerability.

<table>
<thead>
<tr>
<th>Vulnerable C code (unsecure)</th>
<th>Non-Vulnerable C code (more secure)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main(void) { char buff[15]; int pass = 0; printf(&quot;Enter the password:\n&quot;); gets(buff); if(strncmp(buff, &quot;thegeekstuff&quot;) { printf(&quot;Wrong Password\n&quot;); } else { printf(&quot;Correct Password\n&quot;); pass = 1; } if(pass) { printf(&quot;\n<em><strong>Root privileges given to the user</strong></em>\n&quot;); return 0; }</td>
<td>int main(void) { char buff[15]; int pass = 0; printf(&quot;Enter the password:\n&quot;); fgets(buff, sizeof(buff), stdin); if (strncmp(buff, &quot;thegeekstuff&quot;,12)) { printf(&quot;Wrong Password\n&quot;); } else { printf(&quot;Correct Password\n&quot;); pass = 1; } if(pass) { printf(&quot;\n<em><strong>Root privileges given to the user</strong></em>\n&quot;); return 0; }</td>
</tr>
</tbody>
</table>
Chapter 5: Vulnerability Patterns and Detection

5.2.1 The Vulnerability Functions in C Style.

The following explain the insecure pattern features:

(a) Insecure Input Design

gets() or gets(char *s): This function is always vulnerable. The stdio gets() function does not check for buffer length and always results in a vulnerability. gets() doesn't allow you to specify the length of the buffer to store the string in. This would allow people to keep entering data past the end of the buffer. gets() will devour and throw away the newline at the end of the line \n is encountered and An input error occurs.

(b) Insecure Parameters Comparison

Strcmp() or strcmp(const char *s1, const char *s2): This function compares the entire string down to the end without determining the comparison size of the buffer.

5.2.2 Non-Vulnerability Functions in C Style.

The following explain the more secure pattern features:

(a) More Secure Input Design

fgets() or fgets(char *s, int size, FILE *stream): dynamically allocated memory. fgets() reads input and saves to a buffer until: 1) The buffer is being full, 2) \n is encountered, 3) The stream reaches an end-of-file condition, 4) An input error occurs. fgets() allows you to specify a maximum string length. fgets() will store it at the end of the string (space permitting).

(b) More Secure Parameters Comparison

strncmp() or strncmp(const char *s1, const char *s2, size_t n): This function only compares the first n characters of the strings based. Means the size of the comparison already determined in order to allocate place in the memory to avoid any overflow with any other memory places. The above show the example of C language. However we use the disassembler to generate the assembly code as shown below. Reversing basics: understanding the assembly. We disassembled this program, and reviewing the Assembly code generated by the compiler, GCC.
**Table 5-4 Illustrate Disassembly Code of the C Program.**

<table>
<thead>
<tr>
<th>Vulnerability in Assembly</th>
<th>Non-Vulnerability in Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC0: .ascii &quot;Enter the password:\0n&quot;</td>
<td>LC0: .ascii &quot;Enter the password:\0n&quot;</td>
</tr>
<tr>
<td>LC1: .ascii &quot;thegeekstuff\0n&quot;</td>
<td>LC1: .ascii &quot;Wrong Password\0n&quot;</td>
</tr>
<tr>
<td>LC2: .ascii &quot;Wrong Password\0n&quot;</td>
<td>LC2: .ascii &quot;Correct Password\0n&quot;</td>
</tr>
<tr>
<td>LC3: .ascii &quot;Correct Password\0n&quot;</td>
<td>LC3: .ascii &quot;Correct Password\0n&quot;</td>
</tr>
<tr>
<td>LC4: .ascii &quot;\12<em><strong>Root privileges given to the user</strong></em>\0n&quot;</td>
<td>user***\0n&quot;</td>
</tr>
</tbody>
</table>

_main:  
LFB10:  
push ebp  
mov ebp, esp  
and esp, -16  
sub esp, 48  
call ___main  
mov DWORD PTR [esp+44], 0  
mov DWORD PTR [esp], LC0  
call _puts  
lea eax, [esp+29]  
mov DWORD PTR [esp], eax  
call _gets  
mov DWORD PTR [esp+4], OFFSET FLAT:LC1  

OFFSET FLAT:LC0  
call _puts  
lea eax, [esp+29]  
mov DWORD PTR [esp], eax  
call _strcmp  
test eax, eax  
je L2  
mov DWORD PTR [esp], OFFSET FLAT:LC2  
call _puts  
jmp L3  

L2:  
mov DWORD PTR [esp], OFFSET FLAT:LC3  
call _puts  
mov DWORD PTR [esp+44], 1  

L3:  
cmp DWORD PTR [esp+44], 0  
jc L4  
mov DWORD PTR [esp], OFFSET FLAT:LC4  
call _puts  

L4:  
mov eax, 0  
leave  
ret  

LFE10:  
OFFSET FLAT:LC1  
lea eax, [esp+29]  
mov DWORD PTR [esp], eax  
call _strcmp  
test eax, eax  
jc L2  
mov DWORD PTR [esp], OFFSET FLAT:LC2  
call _puts  
jmp L3  

L2:  
mov DWORD PTR [esp], OFFSET FLAT:LC3  
call _puts  
mov DWORD PTR [esp+44], 1  

L3:  
cmp DWORD PTR [esp+44], 0  
jc L4  
mov DWORD PTR [esp], OFFSET FLAT:LC4  
call _puts  

L4:  
mov eax, 0  
leave  
ret  

LFE10:  
OFFSET FLAT:LC1  
lea eax, [esp+29]  
mov DWORD PTR [esp], eax  
call _strcmp  
test eax, eax  
jc L2  
mov DWORD PTR [esp], OFFSET FLAT:LC2  
call _puts  
jmp L3  

L2:  
mov DWORD PTR [esp], OFFSET FLAT:LC3  
call _puts  
mov DWORD PTR [esp+44], 1  

L3:  
cmp DWORD PTR [esp+44], 0  
jc L4  
mov DWORD PTR [esp], OFFSET FLAT:LC4  
call _puts  

L4:  
mov eax, 0  
leave  
ret  

LFE10:  
OFFSET FLAT:LC1  
lea eax, [esp+29]  
mov DWORD PTR [esp], eax  
call _strcmp  
test eax, eax  
jc L2  
mov DWORD PTR [esp], OFFSET FLAT:LC2  
call _puts  
jmp L3  

L2:  
mov DWORD PTR [esp], OFFSET FLAT:LC3  
call _puts  
mov DWORD PTR [esp+44], 1  

L3:  
cmp DWORD PTR [esp+44], 0  
jc L4  
mov DWORD PTR [esp], OFFSET FLAT:LC4  
call _puts  

L4:  
mov eax, 0  
leave  
ret  

LFE10:  
OFFSET FLAT:LC1  
lea eax, [esp+29]  
mov DWORD PTR [esp], eax  
call _strcmp  
test eax, eax  
jc L2  
mov DWORD PTR [esp], OFFSET FLAT:LC2  
call _puts  
jmp L3  

L2:  
mov DWORD PTR [esp], OFFSET FLAT:LC3  
call _puts  
mov DWORD PTR [esp+44], 1  

L3:  
cmp DWORD PTR [esp+44], 0  
jc L4  
mov DWORD PTR [esp], OFFSET FLAT:LC4  
call _puts  

L4:  
mov eax, 0  
leave  
ret  

LFE10: |
Chapter 5: Vulnerability Patterns and Detection

We noticed from the result above that there are some different in code when we disassembly the binary executable files between vulnerability and non-vulnerability with the same code of the program:

Table 5-5 Illustrate the Differences between Vulnerability Pattern and non-Vulnerability Pattern

<table>
<thead>
<tr>
<th>Assembly Pattern of C Style Vulnerable functions</th>
<th>Assembly Pattern of C Style Non-Vulnerable functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>call _puts lea eax, [esp+29] mov DWORD PTR [esp], eax</td>
<td>call _puts mov eax, DWORD PTR _imp__job mov DWORD PTR [esp+8], eax</td>
</tr>
<tr>
<td></td>
<td>mov DWORD PTR [esp+4], 15 lea eax, [esp+29] mov DWORD PTR [esp], eax</td>
</tr>
<tr>
<td>call _gets mov DWORD PTR [esp+4], OFFSET FLAT:LC1 lea eax, [esp+29] mov DWORD PTR [esp], eax</td>
<td>call _fgets mov DWORD PTR [esp+8], 12</td>
</tr>
<tr>
<td></td>
<td>mov DWORD PTR [esp+4], OFFSET FLAT:LC1 lea eax, [esp+29] mov DWORD PTR [esp], eax</td>
</tr>
<tr>
<td>call _strncpy test eax, eax je L2 mov DWORD PTR [esp], OFFSET FLAT:LC2</td>
<td>call _strncpy test eax, eax je L2 mov DWORD PTR [esp], OFFSET FLAT:LC2</td>
</tr>
</tbody>
</table>

(a) If call function (“call _puts”) come straight away before call function (“call _gets”) with missing the following instructions from the call _puts it will cause a buffer overflow vulnerability.

```
mov eax, DWORD PTR _imp__job
mov DWORD PTR [esp+8], eax
mov DWORD PTR [esp+4], 15
```

Listing 5-1 More Secure Code of Assembly

We noticed that the address [esp+8] and [esp+4] have been used in both function “_puts” and “_fgets” means that the entry value already determined in the mentioned address with e size and this help to restrict the capacity of the entry value to avoid the memory leak.

(b) If the register of call function (“call _gets”) have been used in call function (“call _strncpy”) with missing the following instructions from the call _gets it will cause a
buffer overflow vulnerability we noticed that the address [esp+8] have been allocated register “eax” which has been used in three function call “_puts”, “_fgets”, “_strncpy”.

\[
\text{mov DWORD PTR [esp+8], 12}
\]

Listing 5-2 Size Of Variable Determined in the Location means that if we need to do comparison with in input user then we need to check if the size of input have been determined in the input function call “_fgets” and that help to allocated the size in the memory address for the value that we want to compare with it.

5.2.3 Assembly Pattern Instructions of C Style Vulnerable Functions

To understand the assembly, the assembly code generated by the compiler, GCC was reviewed. The binary file disassembler is used for the restoration of the software application code in a format and structure that is readable and understandable to humans. Furthermore, the assembly code file is applied in reverse engineering processes to establish the computer program’s logic flows or its potential vulnerabilities during execution time in real environment. The following example is considered a vulnerable instruction because it is unsecure without determining the addresses for eax and for eax’s size in “_puts” function and also there is no size determined in “_gets” function for “_strcpy” function to help get same size of the value that will compare with.

\[
\begin{align*}
call & \_puts \\
\text{lea} & \text{ eax, [esp+29]} ; \text{(load effective address)} \\
\text{mov} & \text{ DWORD PTR [esp], eax} ; \text{Move value of eax into the address in ESP.} \\
call & \_gets \\
\text{mov} & \text{ DWORD PTR [esp+4], OFFSET FLAT:LC1} ; \text{Place "thegeekstuff\"0" in address [esp+4]} \\
\text{lea} & \text{ eax, [esp+29]} ; \text{(load effective address)} \\
\text{mov} & \text{ DWORD PTR [esp], eax} ; \text{Move value of eax into the address in ESP.} \\
call & \_strcmp \\
\text{test} & \text{ eax, eax} ; \text{Tests if EAX is zero or not. 'je' will jump if}
\end{align*}
\]

Listing 5-3 Disassembly of Flawed gets() and strcmp() Calls in Test Program.

LEA (load effective address) is often used as a "trick" to do certain computations, but that's not its primary purpose. That has the same functionality but requires some bytes more code-space. That's the reason why the compiler emitted a test instead of a compare. The x86 instruction set was designed to support high-level languages like Pascal and C, where arrays especially arrays of ints or small structs are common. Move the 32-bit integer representation of eax into the 4 bytes starting at the address in ESP. In example the test eax, eax will set the zero flag if eax is zero, the sign-flag if the highest bit set and some other flags as well.
Chapter 5: Vulnerability Patterns and Detection

(a) Memory Frame for Vulnerability Assembly Code

During the first function execution esp and ebp have the same value only immediately after
the instruction mov ebp, esp. After that and esp, -16 zeroes the lowest 4 bits (the lowest nibble)
of esp, and esp and ebp diverge, unless the lowest bits of esp were zeroes already. Then sub esp,
48 subtracts 48 from esp and here esp and ebp diverge for sure.

```
push ebp  ; esp = esp - 4; [esp] = ebp.
mov ebp, esp ; ebp = esp. create the stack frame.
and esp, -16 ; make lowest 4 bits of esp zeros, for alignment.
sub esp, 48 ; esp = esp - 48. make space for local variables.
```

Listing 5-4 Created Memory Frame which Caused Vulnerability in Assembly

“Return address” needed in the stack for each call function. [esp+29] not exist so we consider
it as a null value because its outsource. Here we consider that we want to hack the code and if we
put more than 12 characters then it will cause buffer overflow and the address [esp+44] will be
changed from 0 to 1.

![Stack Memory Frame](image)

Figure 5-1: Memory Frame for Vulnerability Assembly Code.
Chapter 5: Vulnerability Patterns and Detection

If more than 12 characters are inserted into the code, then the output will be as shown in the following figure.

Wrong Password
***Root privileges given to the user***

Figure 5-2 Output of the Vulnerable Code in Assembly

5.2.4 Assembly Pattern Instructions of C Style Non-Vulnerable Functions

The following example is considered as a non-vulnerable instruction because it is more secure with determining the addresses for eax and for eax’s size in “_puts” function and also there is size determined in “_fgets” function for “_strncpy” function to help get same size of the value that will compare with.

It was observed that the input placed in register EAX in instruction  eax, DWORD PTR __imp___iob the register placed in address [esp+8] and the determined size of it places in address [esp+4] means there is relation between address [esp+8] and [esp+4] which restrict the input. Moreover, the function “_fgets()” restrict the size of value for the function “_strncpy” when places [esp+8], 12 as the compare value is 12 bytes.

```
call _puts
mov eax, DWORD PTR __imp___iob; Move value of input into the register in EAX
mov DWORD PTR [esp+8], eax ; Move value of eax into the address in ESP+8.
.mov DWORD PTR [esp+4], 15 ; Move value 15 into the address in ESP+4.
lea eax, [esp+29] ; (load effective address)
.mov DWORD PTR [esp], eax ; Move value of eax into the address in ESP.
call _fgets
.mov DWORD PTR [esp+8], 12 ; Move value 12 into the address in ESP+8.
.mov DWORD PTR [esp+4], OFFSET FLAT:LC1 ; Place "thegeekstuff\0" in address [esp+4]
lea eax, [esp+29] ; (load effective address)
.mov DWORD PTR [esp], eax ; Move value of eax into the address in ESP.
call _strncpy
.test eax, eax ; Tests if EAX is zero or not. 'je' will jump if zero.
```

Listing 5-5 Disassembly of more Safe fgets() and strncmp() Calls in Test Program

(a) Memory Frame for Non-Vulnerable Assembly Code

Here we consider that we want to hack the code and if we put more than 12 characters then it will not cause buffer overflow and the address [esp+44] will not be changed from 0 to 1 because the value size been determined in the instruction.

87
Chapter 5: Vulnerability Patterns and Detection

If more than 12 characters are inserted then the output will be as the following:

```
Wrong Password
```

**Listing 5-6 Output of Secure Code in Assembly**

It is concluded that values must be determined in term of the size if there is comparison instruction or outsource input instruction otherwise we consider it as potential vulnerability. It was observed that the call function “`puts`” determine the value of each address and for each function and it is the point where we need to analyses with each function in order to explore if there is problem in the code or not. (For further details see appendix D).

(a) **ASM2WSL**

The ASM2WSL is a tool that performs the translation from Assembly to WSL Action System. This tool was developed and improved upon as part of the overall research to allow for the translation of masm syntax Intel assembly 8086 “X86” 32 bit from GCC. This also include TASM, MASM and NASM based on the manual writing of the implemented example in this...
Chapter 5: Vulnerability Patterns and Detection

chapter. Full details of the codes are provided in Appendix D. The following code is WSL Action System code that has been translated from Assembly using the same example as before. In order to execute the ASM2WSL, it is required to run the “command prompt platform” and then navigate through the path where the source file is from where the command “JAVA asm2wsl filename.asm” can be issued to generate the WSL from the assembly code as illustrated below:

**LISTING 5-7 WSL ACTION SYSTEM**

```assembly
C:\" This file automatically converted from assembler\";
VAR < ax := 0, bx := 0, cx := 0, dx := 0, si := 0, di := 0, bp := 0, sp := 0, cs := 0, ds := 0, esp := 128, ss := 0, ds := 0, ecx := 0, edx := 0, esi := 0, edi := 0, ebp := 0, esp := ARRAY(100), es := 0, ecx := 0, edx := 0, esi := 0, edi := 0, ebp := 0, esp := 0, flag := 0, flag_d := 0, flag_i := 0, flag_r := 0, flag_z := 0, flag_s := 0, flag_p := 0, flag_a := 0, flag_c := 0, overflow := 256, stack := ARRAY(148), t_e_m_p := 0 >:
VAR < skipvar := 0 >:
SKIP;
ACTIONS A_S_start:
A_S_start ==
lc0 := \"Enter the password:\";
lc1 := \"thegeekstuff\";
lc2 := \"Wrong Password\";
lc3 := \"Correct Password\";
lc4 := \"***Root privilege granted to the user***\";
CALL _main; END
_main ==
CALL lfb10; END
lfb10 ==
eax := stack[esp + 29];
stack[esp] := \"! potential Vulnerability \";
CALL _gets;
END
_ends ==
eax := 0;
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
```

WSL Action system

PRINT(stack[esp]);
stack[esp] := eax;
CALL _gets;
C:\\"! potential Vulnerability \";
SKIP
END
.ends ==
eax := 0;
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
eax := stack[esp];
stack[esp] := \"! potential Vulnerability \";
CALL Z;
END
Z ==
```
Chapter 5: Vulnerability Patterns and Detection

In the above code, the Action system of WSL is an assemblage of procedures that are parameter less and are mutually recursive. Here we tried to transform from WSL to WSL by restructuring the code in order to get rid of the action call system, dead code and recursion. By using the transformation method we are refining and simplifying the code to get better understanding level of the code. The first step is to run the start action system and make sure the output of the code has not been changed. Then we apply the transformation to simplify the code based on the required factors. The entire system is then terminated by a special action call labelled as CALL Z, suggesting that Z shouldn’t be used to name an action. When an action system whose execution is caused by every action and leads to a CALL, it is termed a regular action system, sharing similar attributes to a group of labels as well as GOTO function given that there is no action call that can be recalled. Also, within a loop structure, action calls cannot appear. In the framework developed in this work, the aim of the action systems is to enable the translation of codes in machine and spaghetti language into a well-structured WSL. (For further details see appendix D).

(b) Automated Transformations WSL2WSL

This tool was built by Meta_WSL with the view to restructure the WSL Action System to basic WSL code so as to help developers to simplify the code by removing all unnecessary code from the program whilst keeping the semantic of the program the same, without affecting the execution path of the program. Also this helps to avoid the time wasting while running or analysing the code from a different perspective. It also guarantees a better performance and the quality for the code and can assist in detecting vulnerability at a faster rate. In order to implement the transformation it is required to execute the “DOIT.bat” file from command prompt platform as follows:

```
C: cd FME/engine/fermat3
C: doit
```

Listing 5-8 Commands to Run the FermaT Tool

This also set up FermaT to be run from the command line from where the path for which the file to be transformed in entered using the command “wsl transf-min.wsl” followed by “filename.wsl”. All the steps involved in carrying out the full transformation of the program as well as the transformation rules are already described in Chapter 4. (For further details see appendix D).

Transformation tool will generate new file call “filename_t.wsl” and the result of it is as the following.
The generated code looks simple with less number of lines and much smaller than action system and assembly. Also, it functions in the same way as previous examples as shown in Figure D-10. (For further details see appendix D).
5.2.5 Vulnerability Detection level

The vulnerability detection scheme is examined through two stages. First, it analyses the code to identify the vulnerability function and second it analyses the codes in terms of the memory. The two stages have been used in order to ensure the accuracy of the results and to illustrate how vulnerability detection capabilities has been improved based on the extension of WSL using FermaT transformation techniques. Static Impact Analysis depends on:

1. Analysis of source code.
2. Based on assumptions of all possible software runtime.
3. Behaviours Results can include most of the software system in the impact set.

The impact analysis must therefore act recursively looking for relationships that have any of the “dependent artefacts” as sources.

5.2.6 Vulnerability Analyser with Slicing Transformation

In this section taint analysis and slicing method are adopted in order to extract the potential vulnerability from the code. The use of the tool which stems from the current work is vital given that it helps the developer to acknowledge some improvement during migration as to whether the codes is susceptible to vulnerabilities or not. The tool is developed by combining the FermaT Slicing theory with the taint analysis to allow for easy tracking of the vulnerabilities detected. In doing so, the vulnerability associated with the relevant code will be extracted accurately without consuming time, whilst ensuring that the pathway through the execution of the vulnerability is optimal. This will give the developer much more details about the vulnerable code. The idea of the vulnerability analyser is to detect the potential vulnerability by finding the relationship and connection of the vulnerability functions between each other and the relative vulnerability functions to ascertain if they have more than one relationship with other vulnerability functions. By understanding such relationship, the VA can reveal a potential vulnerability within the system. The overall scheme is illustrated in Figure 5-5.
Chapter 5: Vulnerability Patterns and Detection

Figure 5-5: Shows the Slicing Method of the Detection in WSL.

First Taint Analysis ($TA^1$) scan for the entire potential vulnerability pattern as an initial scan to identify pattern function ($P_F$) from the original program save every detect function (F) the information system in the databased and consider it as a first detection result ($R^1$), then the recursive slicing method is applied for every single potential vulnerability function to extract all the related functions for every single vulnerability function separately in order to follow the vulnerability relations functions easily. Thereafter, a second taint analysis ($TA^2$) is performed again as a second scan to identify final detection of every relation vulnerability function with other vulnerability functions. Save the final detection result ($R^2$) as an information system compare and match the number of vulnerability ($M^n$) with the first result and remove any potential vulnerability that has no more than one. This method can easily track, extract and detect the vulnerability accurately.

\[
D_F = \left\{
\begin{array}{l}
TA^1_F = P_F \\
TA^2_F \geq TA^2_F \\
TA^2_F \geq M^n \\
R^1 - R^2
\end{array}
\right.
\]

In Chapter 4 the concept of inter-procedural taint analysis was introduced for the identification of variables and functions that have become 'tainted' with user controllable input. The sources are traced to identify possible vulnerable functions termed a 'sink'. If the tainted variable gets passed to a sink without an initial screening and sanitisation, it is flagged as vulnerability. A result of the sample code that was the output of the ASM2WSL after the code has been structured is presented as shown below.
### Table 5-6 Detected Vulnerability Pattern in the Tested WSL Program

<table>
<thead>
<tr>
<th>Taint Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>******************</td>
</tr>
<tr>
<td><strong>Vulnerability type:</strong> Buffer Overflow (String Copy and Read Bugs with String Compare Bug)</td>
</tr>
<tr>
<td><strong>Function/Pattern of the potential vulnerabilities</strong></td>
</tr>
<tr>
<td>Line 12: stack[44] := @Read_Line(Standard_Input_Port);</td>
</tr>
<tr>
<td>Line 15: IF stack[44] = &quot;thegeekstuff&quot;</td>
</tr>
<tr>
<td>******************</td>
</tr>
<tr>
<td><strong>Classification of the potential vulnerabilities</strong></td>
</tr>
<tr>
<td>// Variable (stack[44]) and (stack[15]) connect to the following:</td>
</tr>
<tr>
<td>Line 1: ARRAY</td>
</tr>
<tr>
<td>Line 12: Read Bug</td>
</tr>
<tr>
<td>Line 14: String Copy Bug</td>
</tr>
<tr>
<td>Line 15: String Compare Bug</td>
</tr>
<tr>
<td>******************</td>
</tr>
</tbody>
</table>

**Note:** If the same variable has “ARRAY” connect with read bug such as “Variable:= @Read_Line” and “Variable:= Variable” and compare bug such as “IF statement” then it indicate the potential vulnerability.

As indicated above, if `@Read_Line` is a tainting function, then `stack[44]` line 12 will be marked as tainted. The Slicing algorithm as illustrated in Figure 5-6 will make use of `stack[44]` in the expression on whole program and will taint its result, which is stored in different variable. On different lines a value that is tainted is used in a call to “IF” statement, a function that can be potentially vulnerable. Such form of vulnerability has the potential to be appropriately established and reported based on the tool developed as part of this research.

**Figure 5-6:** Shows the Detection of Vulnerable Code.
In order to use the slicing in the FME, the Abstract Syntax Tree needed to be used as it helps to click on the node where the potential vulnerability is. The suspicious variables points are then sliced and analysed. Figure D-13 (in Appendix D) shows the Abstract Syntax Tree (AST) displaying the hierarchy, top-bottom, where the potential vulnerability is positioned based on the following command.

\begin{verbatim}
(@Print_WSL (@I) "");
(@Print_WSL (@Program) "");
\end{verbatim}

Based on the extraction of abstract syntax trees from the code and the determination of structural patterns in these trees, each function in the code can be labelled as a mixture of the extracted patterns. This representation allows for the decomposition of a known vulnerability and allows for its extrapolation to a code base. This ensures that functions with similar pattern of flaw can be recognised by the systems analyst.

In order to slice each of the vulnerabilities assignments, it is required to select each of the assignment/item in AST in order to highlight them within the code by entering (@Posn) at the console to show the position of that item. A position as used here is a sequence of integers specifying the component numbers from the root of the program. For example, an array assignment such as (stack [15]: = stack [44] ;) updates the value stored in stack: so stack is assigned. But it only updates array element 15 of stack: the other array elements are assigned their original values. As such, the assignment statement uses the current value of stack to generate the new value of stack. So stack is both used and assigned in the statement. In order to apply the slicing to the example, it requires all assignments to be simple variables.

Syntactic slicing treats stack [15] as a separate variable in stack. The selected item for the transformation is the block of code which we are slicing over: this is usually the whole program. To specify the slicing point, one can pass one or more variables as data to the transformation and optionally pass a position to specify the slice point.

Each position of selected item or assignment will determine the variables that are required to be sliced and then apply the FermaT syntactic slicing transformation to delete unreachable code whilst keeping all the statements that affects the value of variable. Table 5-7 shows the position point of each assignment statement as the following:

**Table 5-7: The Position Point of each Assignment Statement.**

<table>
<thead>
<tr>
<th>Potential Vulnerability</th>
<th>Position Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack[44] := @Read_Line(Standard_Input_Port);</td>
<td>(5)</td>
</tr>
<tr>
<td>stack[15] := stack[44];</td>
<td>(7)</td>
</tr>
<tr>
<td>stack[44] = &quot;thegeekstuff&quot;</td>
<td>(8 1 1)</td>
</tr>
</tbody>
</table>
Note that, unlike a backwards slice, a forwards slice is not necessarily a valid program. In this case, the forwards slice may not terminate if executed. To slice on stack [15] it is probably simplest to rename all the array references as separate variables, eg stack_15 etc.

```
stack_59 := 0;
stack_15 := "Enter the password:";
PRINT(stack_15);
stack_15 := stack_44;
stack_44 := @Read_Line(Standard_Input_Port);
stack_19 := "thegeekstuff";
stack_15 := stack_44;
IF stack_44 = "thegeekstuff" THEN
  stack_15 := "Correct Password";
  PRINT(stack_15);
  stack_59 := 1;
  stack_15 := "***Root privileges given to the user***";
  PRINT(stack_15);
14(VAR)
ELSIF SLENGTH(stack_44) > 12 THEN
  stack_15 := "Correct Password";
  PRINT(stack_15);
  stack_59 := 1;
  stack_15 := "***Root privileges given to the user***";
  PRINT(stack_15);
14(VAR)
ELSE
  stack_15 := "Wrong Password";
  PRINT(stack_15);
14(VAR)
FI
```

After renaming the stack variables and removing the VAR.

**Listing 5-10 Renamed Array References as Separate Variables**

To slice on variable stack_15 at this point, pass this data (variable name with @ symbol and its location in the code for example (Variable@(position))) as an argument to Syntactic_Slice: stack_15@(7). The slicing criterion passed to the transformation must be a variable and an optional position. A syntactic slice produces:

```
stack_44 := @Read_Line(Standard_Input_Port);
stack_15 := stack_44
```

**Listing 5-11 Detected the Overwritten Variables by Slicing with Pointed Taint Position**

The stack_15 is changed by statement stack_44 based on the value of Standard_Input_Port. Standard_Input_Port is modified elsewhere, so other code is included indirectly via dataflow analysis, which is stack_15 := stack_44. Means if we execute the original program and execute the slice, then stack_15 gets the same value in both executions. So the slice, which is just the couple statements stack_44:= @Read_Line (Standard_Input_ Port) and stack_15 := stack_44,
Chapter 5: Vulnerability Patterns and Detection

Listing 5-12 Detected the Relevant Code by Using Slicing Transformation with Taint Analysis

At this stage, slicing is applied to the original code. A syntactic slicer that is conditioned, we have applied the syntactic slicing for the both positions stack_44(5) and stack_15(7), also we have applied the syntactic slicing without positons in order to see the different of the output to trace the details of the result as shown below.

Table 5-8: Detecting Vulnerable Code with Syntactic Slicing Method.
Chapter 5: Vulnerability Patterns and Detection

This syntactic slice deleted some of the unnecessary statements given that the codes will never be executed as indicated by the functions written. In addition, the group of vulnerability per each slicing of the variable from different slice method. Figure D-14 (in Appendix D) shows the syntactic slice that was applied within the overall codes. Outputs based on semantic slicing is illustrated in Figure D-15.

Table 5-9 Detecting Method with Semantic Slicing.

<table>
<thead>
<tr>
<th>Semantic Slicing (stack_44)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 stack_44 := @Read_Line(Standard_Input_Port)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semantic Slicing (stack_15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 IF @Read_Line(Standard_Input_Port) = &quot;thegeekstuff&quot; THEN</td>
</tr>
<tr>
<td>2 stack_15 := &quot;<em><strong>Root privileges given to the user</strong></em>&quot;</td>
</tr>
<tr>
<td>3 ELSIF SLENGTH(@Read_Line(Standard_Input_Port)) &lt;= 12 THEN</td>
</tr>
<tr>
<td>4 stack_15 := &quot;Wrong Password&quot;</td>
</tr>
<tr>
<td>5 ELSE</td>
</tr>
<tr>
<td>6 stack_15 := &quot;<em><strong>Root privileges given to the user</strong></em>&quot;</td>
</tr>
<tr>
<td>7 FI</td>
</tr>
</tbody>
</table>

This semantic slice evaluates the expressions, ensure the removal of redundant variables and assignments, and identifies which path it will always take in the IF conditions to populate the PRINT statement with the classification, through the evaluation of the IF conditions which helps in the overall taint analysis to allow easy recognition of pattern. Figure 5-7 shows the semantics of the slice that was applied within the overall to development. After obtaining a set of suspicious fragments, it becomes easier for the taint analysis to ascertain the pattern of vulnerability which have been extracted from different slicing methods based on syntax and semantics. Both slicing methods yielded slightly different result given that stack_44 is the source of the main vulnerable and is considered an untrusted source because it derives its source from outside the function: (@Read_Line(Standard_Input_Port) which was used in stack_44 The call graph tracks the flow of values between procedures/functions and the output from syntax show that the call function of @Read_ has been used one time, however the semantic shows two times. This second iteration of taint analysis will help to easily find the vulnerability for each slicing recursively such that the result indicates three vulnerable have been found in the vulnerability analyser. Sematic slicing stack_15 showed the common pattern that appeared in both slicing transformation which indicated the suspicious positon in the code and help the taint analysis to detect the hidden vulnerabilities.

The semantic slicing generates concise and smaller compared to syntactic slicing given that the program syntax can be re-written in a way to exhibit behaviour as the syntactically sliced
Chapter 5: Vulnerability Patterns and Detection

program. However, experience in terms of the domain knowledge as well as an understanding of the program is a mandatory requirement for slicing procedures based on syntax. Furthermore, the syntactic slicing is confined to the maintenance of the program sequence and does not involve the evaluation of the expressions to make adjustments to the values, explaining while its program length is still longer compared to semantic slicing. Figure 5-7 gives a pictorial example of result outputs based on slicing.

![Figure 5-7: Slicing Result Output.](image)

When the number of suspicious fragments of Intermediate Representation (IR) is very large, the detection analysis policy will be important. Accordingly, in the overall tool development, a detection analysis policy which takes into consideration the following factors was designed:

1. Function properties which states whether the function is an export function or not, and whether the function is from a referenced library function or user developed, etc.
2. The level of vulnerability potential.
3. Functional dependency between relations. For example, mechanism such as multi-thread can be adopted to simultaneously initiate multiple checking procedures. When a checking mission is accomplished, the results will be revealed in a friendly manner to the developer. Table D-9 shows the figures on vulnerability slicing to detect the vulnerabilities. (For further details see Appendix D).

5.2.7 Memory Analyser with Static Single Assignment

Bound checking is a technique of detecting whether a variable is within a defined bound before it is employed. It finds application is ensuring that a number fits into a range checking or that a given variable is adopted as an array index within the bounds of an array (i.e. index checking). In the tool development, Boundary Checking of Value Range allows for the checking of the size of the value that affects the original value of the variable that has been overwritten. The application of taint analysis to a vulnerable program allows for the discovery of execution paths where information or data is read from a source that is adjudged untrusted and used inappropriately. Another attributes of vulnerability which the current model handles pertains to
Chapter 5: Vulnerability Patterns and Detection

instances where data not sufficiently validated. In this case, the first step is to separate the variable as a single mean any overwritten happen to the variable will consider as a new variable then we save the updated version of each actual variable as history of the variable to know in which part been overwritten. To do this, a transformation technique based on FermaT based on formal method known as “Static Single Assignment” (SSA) form was adopted. SSA can help in detecting the actual vulnerability that has been overwritten. A combination of rules of boundary checking with the FermaT SSA Transformation, help in the tracking of the weighted value ranges of variables, either symbolic or numeric, through a program. Branch prediction is then carried through consultation with the value range of the correct variable. The outputs of the results from slicing and SSA transformation are illustrated in Table 5-10 below.

Table 5-10 VA Output an SSA Process for Detecting Vulnerability

<table>
<thead>
<tr>
<th>VA Output</th>
<th>SSA Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack_44 := @Read_Line(Standard_Input_Port);</td>
<td>stack_44_1 := @Read_Line(Standard_Input_Port__0);</td>
</tr>
<tr>
<td>stack_15 := stack_44;</td>
<td>stack_15__1 := stack_44__1;</td>
</tr>
<tr>
<td>IF @Read_Line(Standard_Input_Port) =</td>
<td>IF @Read_Line(Standard_Input_Port__0) =</td>
</tr>
<tr>
<td>&quot;thegekstuff&quot; THEN</td>
<td>&quot;thegekstuff&quot; THEN</td>
</tr>
<tr>
<td>stack_15 := &quot;***Root privileges given to the</td>
<td>stack_15__2 := &quot;***Root privileges given to the</td>
</tr>
<tr>
<td>user***&quot;</td>
<td>user***&quot;</td>
</tr>
<tr>
<td>ELSIF</td>
<td>ELSIF</td>
</tr>
<tr>
<td>SLENGTH(@Read_Line(Standard_Input_Port))</td>
<td>SLENGTH(@Read_Line(Standard_Input_Port__0))</td>
</tr>
<tr>
<td>&lt;= 12 THEN</td>
<td>&lt;= 12 THEN</td>
</tr>
<tr>
<td>stack_15 := &quot;Wrong Password&quot;</td>
<td>stack_15__3 := &quot;Wrong Password&quot;</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>stack_15 := &quot;***Root privileges given to the</td>
<td>stack_15__4 := &quot;***Root privileges given to the</td>
</tr>
<tr>
<td>user***&quot;</td>
<td>user***&quot;</td>
</tr>
<tr>
<td>FI</td>
<td>FI</td>
</tr>
</tbody>
</table>

Vulnerability Conditions that been used in the example

1. $\infty \leq x.Size$
   Input = unknown
2. $Src_VarSize \leq Dst.VarMaxSize$
3. $Dest_Max_buff == Src_Var_Size$
   Dest_Value = Src_Value
   Index_VarSrc1 = (VarSrc2.n_length)

This is stored internally as a parse tree, as illustrated in Figure D-16 in Appendix D.

As indicated in each node in the tree records the specific type of the node, the value stored at that node (<> represents the null or empty value), and the sequence of components of the node.

As shown in the above example that the Stack_44_1 and stack_15_1 are the only two variables that have been affected, so in this case the boundary checking will analyse these variable to check for the possibility of a potential vulnerability. Similar to the example before, stack_44 will be
Chapter 5: Vulnerability Patterns and Detection

flagged as tainted. The process for taint analysis in the tool development reports three times uses of stack_44 in the calls on lines 0, 1 and 3 as vulnerable. This is accurate for the call on line 0, because stack_44 can be arbitrarily large. Also, on line 2 a true positive exist which is most of tool having false positive. The IF statement makes certain that the value of stack_44 < the size of the buffer destination, which makes the stack memory call safe. The method illustrated in the current work has been built on algorithm and rules that have been extensively explained Chapter Four. So with the above example, the bound checking tool will consider that the stack_44 is supposed to be no longer than 12 as the comparison string in IF statement show a length of 12 and because the unknown input then the system will send a warning to the alarm unit as the value range has not been checked, so in this case the tendency of a potential vulnerability occurring becomes lesser (For further details see Appendix D).

Vulnerability Analyser alone is not enough to differentiate between the safe and vulnerable use of data that are untrusted in the above example. An analysis that considers the effect of the IF statement and ascertain the potential values of the stack_44 variables in the THEN and ELSE branches is therefore required. It is mode of operation is based on the SSA form of a program, representing each definition of a variable as a separate form of it. This provides the algorithm with the capability to differentiate between the different ranges of stack_44 and stack_15 in each branches of the IF statement. The computation of range of possible values for each variable, helps to check if the variable will affect the stack memory or not. As mentioned in Chapter Four, the algorithm of the boundary checking has the capability to differentiate between the different ranges of stack_44. A summary of the approach for boundary checking for vulnerability detection is provided in Chapter Four. Slicing and taint analysis help in detecting the suspicious variable and using the value range technique with single static assignment to determine a range for each suspicious variable. (For further details see Appendix D)

![Figure 5-8: Result of the Memory Analyser](image)

101
Chapter 5: Vulnerability Patterns and Detection

The stack was built as an array “set/list” to insert the value of each variable before and after in order to detect the overwritten size of each new value or updated value. Figure 5-8 shows that the vulnerability has been detected as the variable size unknown. The memory analyzer will also create the multi array which represent the stack of the memory (bound checking (2, variable value)) as the first array will place the initial value and the second array will place the size of it and when the new value is placed, it will update the element, the analysis rules will then detect these changes and will send the unit alarm if the vulnerability conditions is broken for whatever reason.

5.2.8 Decision Maker of Vulnerability Detection

The decision maker is employed to get the result from both vulnerability analyser and memory analyser and if one of them results in vulnerability then the output result reports a warning and if both analyser detect vulnerability, then a vulnerability alert is signalled. Each of the Vulnerability functions was implemented within the overall tool using if-else statement. The report of the above example is:

```
Vulnerability Type: Buffer Overflow
******************************************************************************
Vulnerability Analyser result
******************************************************************************
Vulnerability analyser: Vulnerability Detected
Number of Affected lines: 3
Number of Vulnerability: 1
******************************************************************************
******************************************************************************
Memory Analyser result
******************************************************************************
Memory analyser: Vulnerability Warning
Number of Affected lines: 2
Number of Vulnerability: 1
******************************************************************************
Decision Maker : Vulnerability Detected
```

Figure 5-9: Validation Code Implemented via FME Program.

5.3 Summary

In this chapter, the modus operandi of the of the overall vulnerability detection tool was developed as part of the current work is presented. The pilot study presented forms the basis of how the final research problem to be addressed in this work emerge, by providing a deeper understanding of the overall research problem and establish a focus for the current work. Also
Chapter 5: Vulnerability Patterns and Detection

Presented is the development of implementation of prototypes for vulnerability detection. Within the overall tool, a multi-method approach with different verification stages was adopted with the view to increase the accuracy of vulnerability detection. For instance, taint analysis was used to determine the functions that contain vulnerability while slicing techniques was used for the simplification and extraction of codes that are susceptible to vulnerability. A combination of taint analysis and slicing helps in ascertaining the accuracy of the vulnerability detection. Furthermore, bound checking was adopted to analyse the variable range value that has been affected by other variables. Given this multi-method approach, it becomes relatively easy to detect many forms of vulnerabilities with slight variations, whilst guaranteeing the security of the overall components within the tool.

Through the use of different examples, it was demonstrated in this chapter that transformation techniques help in isolation and conversion without affecting the entire codes. At the same time, it was highlighted that the use of formal methods and intermediate language can facilitate the detection of vulnerability codes, without causing any damaging effects to the entire program codes. The use of FermaT (WSL) toolkits, especially the meta-WSL provided a solid foundation for the development of transformation techniques embedded in the tool development in this work. For example, it helps in the auditing and improvement of the quality of the codes whilst maintaining the semantics that are equally useful for the construction of memory checking transformations. Overall, the implementation strategy for the vulnerability detection system based on extended WSL was presented in this chapter. SSA and slicing techniques form the core of the framework which was used for the vulnerability detection in this work. The overall work provides further useful insights into the science of vulnerability detection through the use of taint analysis, transformation techniques, value range analysis and vulnerability reporting.
Chapter 6 Prototype Tool Support

Objectives

- To discuss WSL and the tools that are needed to enable it for effective vulnerabilities detection.
- To provide a description the architecture of prototype tool environment.
Chapter 6: Prototype Tool Support

In this Chapter, an overview of the inherent tools within the overall vulnerability detection framework to support the potential users is presented. Such tools as described in this section can be regarded as the documentation for the tool. The documentation details the description of each of the building blocks (e.g. tools and equipment used in the construction of the model), programming languages used, operating systems, transformation techniques as well as the computational frameworks. This allows the users to trust the model more by providing with a detailed understanding of the mechanics of the vulnerability detection and how such vulnerability detection functions are carried out.

6.1 Support Tool (Equipment)

For Vulnerability Detection (VD) mechanism that is in line with FME, the development of a tool support is essential. This section introduces a set of prototype tools, which were developed to provide help in detecting the vulnerability from source code. The related tools used in VD approach are also discussed. All these tools were designed by the current researcher. Automation is a goal of tools, but with the understanding that human intervention is crucial in filling the gap of different abstraction levels, these tools can only support VD approach semi-automatically. VD with certain capabilities are essential for detecting the vulnerability for assembly (low level language); the VD is equipped with highly advanced formal transformation and has been adopted by major VD including translation, transformation, taint analysis and value range analysis.

6.2 Equipment

The VD alone is not sufficient to produce an accurate evaluation of the proposed method algorithm. Hence, there is a need for another source of data that is more accurate than the WSL code to act as a true reference. The true reference C language vulnerability was obtained from the different source and has been tested to ensure there is no vulnerability in the code. To achieve this aim, the following tools as detailed in section 6.3 have been developed.

6.3 Developed Tools for Vulnerability Detection

(a) C2WSL

This tool translates the C code to WSL and was built based on C++ language according to the WSL library. This tool is needed in order to be able to translate the existing vulnerability in C to WSL. This help to establish how the pattern of the vulnerability look like in WSL and also help developing the ASM2WSL by checking the similarity of the translation output. Accordingly,
Chapter 6: Prototype Tool Support

for this tool it is required to copy and paste the code that are to be translated into a specific file through an execution file to generate the new file with the WSL.

Figure 6-1 C2WSL Translation Tool

(b) ASM2WSL Action System

This tool is able to translate the assembly X86 codes to WSL Action System and it is considered an integral part of the proposed method in this research. The tool was built using Java and the translation needed to be able to automate the translation for implementing more examples. The translation method is based on the output of the C2WSL and because not everything in assembly exist in C language. So for this tool we need to copy and paste the code we want to translate in a specific file then we click the execution file to generate the new file with the WSL.
The following is the instruction to run the program

PATH: src file
JAVA asm2wsl filename.asm

Listing 6-1 Command to Run the Translation Tool

Examples of how the translator works is depicted below

Table 6-1 Example of Translation Command from ASM to WSL

<table>
<thead>
<tr>
<th>Assembly</th>
<th>WSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ax, ad</td>
<td>ax := dx</td>
</tr>
<tr>
<td>xchg ax, dx</td>
<td>&lt;ax := dx, dx := ax&gt;</td>
</tr>
<tr>
<td>add dx, ax</td>
<td>overflow := 65536;</td>
</tr>
<tr>
<td></td>
<td>dx := dx + ax;</td>
</tr>
<tr>
<td></td>
<td>IF dx &gt;= overflow THEN</td>
</tr>
<tr>
<td></td>
<td>dx := dx MOD overflow;</td>
</tr>
<tr>
<td></td>
<td>flag_o := 1; flag_c := 1;</td>
</tr>
<tr>
<td></td>
<td>ELSE flag_o := 0; flag_c := 0;</td>
</tr>
<tr>
<td></td>
<td>FI;</td>
</tr>
<tr>
<td>mov ah, n</td>
<td>t_e_m_p := n;</td>
</tr>
<tr>
<td></td>
<td>ax := (ax MOD 256) + t_e_m_p * 256;</td>
</tr>
<tr>
<td>add al, 12</td>
<td>overflow := 256;</td>
</tr>
<tr>
<td></td>
<td>t_e_m_p := (ax MOD 255) + 12;</td>
</tr>
<tr>
<td></td>
<td>IF t_e_m_p &gt;= overflow THEN</td>
</tr>
<tr>
<td></td>
<td>t_e_m_p := t_e_m_p MOD overflow;</td>
</tr>
<tr>
<td></td>
<td>flag_o := 1; flag_c := 1;</td>
</tr>
<tr>
<td></td>
<td>ELSE flag_o := 0; flag_c := 0;</td>
</tr>
<tr>
<td></td>
<td>FI;</td>
</tr>
<tr>
<td></td>
<td>ax := (ax DIV 256) * 256 + t_e_m_p;</td>
</tr>
<tr>
<td>print_str x</td>
<td>PRINT(s);</td>
</tr>
<tr>
<td>print_num x</td>
<td></td>
</tr>
<tr>
<td>read_str x</td>
<td>x := @Read_Line(Standard_Input_Port);</td>
</tr>
<tr>
<td>read_num x</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6: Prototype Tool Support

(c) WSL2WSL

The WSL2WSL tool was built to restructure the output code that has been translated from assembly to WSL in order to simplify the code and to reuse all the dead code to provide a better understanding. The tool was constructed through the Meta-WSL which help to automate the transformations steps based on the requirement to be achieved. To run the program the DOIT.bat from command line window must be executed based on the following.

```
C: cd FME/engine/fermat3
C: doit
```

Listing 6-2 Command to Run the Transformation Engine

To set up FermaT to run from the command line. Then open the command prompt window and enter the path where the files you need to transform then write the following command: “wsl transf-min.wsl” then It will ask you for a file to transform for example “Filename.WSL” then the program will generate new file call (Filename_T.WSL)

Figure 6-3 Transformation Tool

6.3.1 Binary Code Vulnerability Detection Tools

This tool is divided into components (vulnerability analyser and memory analyser). This tool is considered as an extension to FermaT and WSL and helps in detecting vulnerability. This tool was completely built using C++ programming language.
(a) Vulnerability Analyser

This tool helps to detect the call functions of the vulnerability in order to slice the code by using the FermaT transformation. This tool was constructed based on C++ it meant to be integrated with the transformation for immediate extraction of the related code of the vulnerability. Here we need to copy and paste the code we want to analyse in a specific file then we click the execution file to generate the new file with the WSL vulnerability report.

Figure 6-5 Phase of Vulnerability Analyser Tool
Chapter 6: Prototype Tool Support

(b) Memory Analyser

This tool was built as a bound checking facility for analysing the value range of the code been sliced. We use this method to increase the verifying of the existence of the vulnerability. This tool in order to be much more accurate we use also a transformation call “static single assignment” this will help the analysis to analyse the version of the variable that been affected by the vulnerability. Here we need to copy and paste the code we want to analyse in a specific file then we click the execution file to generate the new file with the WSL vulnerability report.

Figure 6-6 Phase of Memory Analyser Tool

(c) Decision Maker

Here will we take the result from the both tool and generate the final report for the developer.

Figure 6-7 Phase of Decision Maker Tool
6.3.2 Other Related Tools

In this section, an introduction to some supporting tools for vulnerability detection in the overall tool development is provided. Although these tools have not been integrated into the VD, but the compatible features allows for the importation of the analysis results of existing systems into these tools for further processing. Example of some of the supporting tools are presented below.

(a) Intel Window Disassembler GCC: (GNU)

Intel Windows 64 bit was used to disassemble the C exe binary file to the Assembly for reverse engineering purposes. This would help to ensure that the vulnerabilities within assembly code can be detected, providing a basis for the validation of the current tool in terms of its vulnerability detection capabilities.

(b) De-compiler REDEC Tool

This code helps to decompile the code by providing more option such as control flow diagram and other related functions. The decompiling is needed to convert the binary file to assembly file so that the translation from assembly to WSL becomes possible. Binary file was used based on the assumption that even if the source code is not available it is still possible to detect vulnerabilities based on the methods proposed in this research.

Figure 6-8 De-compilation of REDEC Tool
Chapter 6: Prototype Tool Support

(c) FermaT Transformation Engine

Transformation Engine, which supports the automatic transformation of programs and models whilst maintaining the integrity of certain properties, is at the functional heart of the overall tool development. Transformation Engine transforms the system based on transformation rules, depending on matching detection. For instance, if the inputs are matched with predefined pattern, the system will be rewritten according to the transformation rules. Accordingly, the transformation engine as adopted in the current research has improved the overall capability of vulnerability detection in low level language upon which legacy system operates.

Figure 6-9 FME Platform Tool.

(d) Visualisation Engine

Visualisation is the main requirement of a presentation tool and it is very important for program comprehension in software re-engineering. To this end, STRL Visualisation Engine (SVE) was developed as a software package, and provides an easy-to-use API to create and present the diagrams for users. All graphic related work could be encapsulated within the SVE so that the developers can create and browse through large scale diagrams easily without worrying about the graphical details.

The SVE consists three main parts which can be extended or modified separately:

- Abstraction components to define a graph mathematically,
- Graphic components for display and navigation, and
Chapter 6: Prototype Tool Support

- Event handling components to define the behaviour of a graph.

**Figure 6-10 Visualisation Engine Tool**

(e) Metrics

A number of functions are provided which measure the complexity of a given WSL item according to a variety of metrics. These help to monitor the changes and where needed to simplify the code.

**Figure 6-11 Metrics Tool of WSL**
Chapter 6: Prototype Tool Support

6.4 Summary

As part of the overall process of developing a new tool, it is important that trust is built into the model for easy utility. Accordingly, it is important to have a proper documentation where all aspects of the developmental steps of the tool such as hardware requirements, software integration and compatibility, migration from one programming language to another one, the transformation engines and techniques used are provided. This is the focus of this chapter based on the overall content presented. The provision of such documentation provides the user with an avenue to look into when seeking an understanding of how various components combine together to aid vulnerability detection in binary executable files.
Chapter 7 Experiments, Analysis and Evaluation

Objectives

- Highlight the experiments conducted to test the prototype implementation presented in Chapter 5.
- Discuss the results of vulnerability detection prototype.
- Evaluate the research that has been explored in this thesis.
- Provide a discussion of the dataset selection and vulnerabilities detection.
- Discuss the theory validation of this research and how it can be met.
- Provide the samples used in this research to analyse normal and vulnerable processes.
Chapter 7: Experiments, Analysis and Evaluation

Vulnerability detection has been discussed in many studies, with most of them covering issues that pertains to software developed using high level programming language. In the current research, the focus is on vulnerability detection in binary executable files using intermediate level language as an intermediary platform, with the view to improve the overall mechanism for detecting vulnerabilities in terms of the accuracy and efficiency. A number of investigations have been carried out to examine the theory of this research. The comprehensive attempt of a number of systems vulnerability mechanisms was highlighted in Chapter 5. The proposed method presented in this work has been tested with 80 examples, yielding promising results in comparison to related works by other researchers. Overall, the method proposed in the current research makes the detection and tracing of vulnerability easier and will be useful for software developers and maintainers.

The usability of the framework developed as demonstrated by a number of examples will be examined in this chapter and different factors and attributes such as false positives, false negatives and true positive and true negative will equally be examined. Overall, this chapter examines the overall results emanating from the framework including vulnerabilities detection for the vulnerability and normal process results. It then evaluates the approaches adopted during the model development based on the research hypotheses and the evaluation criteria namely detection of known and unknown vulnerability and false positive production.

7.1 Verification and Validation of the Model

This entails the overall evaluation of the model framework based on verification and validation strategies employed in this research. Verification entails checking whether the actual model developed is indeed a true representation of the actual purpose it was designed for. On the other hand, validation refers to checking the model to confirm that it actually represents the main concept that is being modelled (Miser, 1997; Miser, 1985; Miser and Quade, 1985). Verification ensures that the system or model is constructed in a right manner and validation entails constructing the right system (O’Keefe et al., 1986). Verification has to do with the logical correctness of the knowledge domain of the DSS, but the logic of the knowledge domain may be correct but not necessarily valid. Hence, validation pertains to how well a model follows what is being modelled (De Kock, 2005; Borenstein, 1998). Validation is concerned with attributes such as data inputs, knowledge base (i.e. concepts and relations), reasoning (i.e. strategies), and results (i.e. conclusions). In this work, the verification and validation approaches integrates information derived from the model with strategies described in chapters Four and Five for detecting vulnerabilities in low level programming language such as assembly language programming. It
Chapter 7: Experiments, Analysis and Evaluation

involves key steps including requirement analysis and specification and implementation and testing strategies. A summary of the steps and strategies involved for the validation and verification of the vulnerability detection model developed in this work is provided in Table E-1, Appendix E.

7.2 Experimental Procedure of the Vulnerability Detection Tool

Three sets of experiments were performed to verify and validate the results of the vulnerability detection from the model. The first experiment was conducted to check for non-vulnerability aspects of the system. The second experiment was performed based on different vulnerability examples and the ability of the system to detect it. The third experiment was performed with unknown examples of vulnerabilities, to ascertain whether the model can detect vulnerabilities without false negatives and whether it can distinguish between vulnerabilities and normal processes by not producing false positives. The experiments were conducted using 80 examples with different codes of vulnerability and non-vulnerability and the results were adequately verified and validated.

To ensure that the precision of the detection mechanism, 20 random samples were used to verify detection alarm and false alarm. This was carried out by gathering all possible conditions under which vulnerabilities can be detected. These are then weighted by using iteration method with the view to measure the risk level associated with each vulnerability detected. To detect vulnerability in assembly language potential vulnerabilities pattern were examined using WSL and each function is assigned a weight until an output is realised as summarized in Table 7-1.

Table 7-1: Weight for Each Pattern.

<table>
<thead>
<tr>
<th>Vulnerability Pattern</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output / PRINT (Src)</td>
<td>LOW</td>
</tr>
<tr>
<td>Src[ ] := Variable</td>
<td></td>
</tr>
<tr>
<td>Chunks / Stack [ ]</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>Memory Frame / Array( )</td>
<td></td>
</tr>
<tr>
<td>Read input / @Read_Line</td>
<td></td>
</tr>
<tr>
<td>Src[ ] := SLENGTH(Variable)</td>
<td></td>
</tr>
<tr>
<td>If statement</td>
<td></td>
</tr>
<tr>
<td>SLENGTH(Dst) = SLENGTH(Src)</td>
<td>HIGH</td>
</tr>
<tr>
<td>Det[ ]:= Src[ ]</td>
<td></td>
</tr>
<tr>
<td>Copy String / Copy Variable</td>
<td></td>
</tr>
<tr>
<td>Dst := Src</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 7: Experiments, Analysis and Evaluation

7.3 Vulnerability Conditions

The vulnerability conditions are the patterns that help to explore the weaknesses of the codes within the WSL framework. Each rule is assigned a weight so that the associated risk can be examined, and aid the decision making of the developer. These weights have been determined based on the samples applied as shown in Table 7-2 and plotted in Figure 7-1.

**Table 7-2: VCs Weight for Detecting Vulnerable Code.**

<table>
<thead>
<tr>
<th>No.</th>
<th>Vulnerability Pattern</th>
<th>Weight</th>
<th>Vulnerability Condition(VC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Copy String / Copy Variable</td>
<td>80%</td>
<td>Dest := Src&lt;br&gt;Src_VarSize ≤ Dest_VarMaxSize</td>
</tr>
<tr>
<td>2</td>
<td>Output / PRINT (Src)</td>
<td>34%</td>
<td>Prnt_Src.VarSize ≤ prnt_Dest.Size</td>
</tr>
<tr>
<td>3</td>
<td>Dest[ ] := Src[ ]</td>
<td>85%</td>
<td>Input = known&lt;br&gt;Src_Siz(&quot;Input&quot;) ≤ Dest_MaxSize</td>
</tr>
<tr>
<td>4</td>
<td>Src[ ] := Variable</td>
<td>67%</td>
<td>Buff_Size + Var_Str.Size ≤ Buff_MaxSize - 1</td>
</tr>
<tr>
<td>5</td>
<td>Chunks / Stack [ ]</td>
<td>63%</td>
<td>Block_Memory_Size &gt; ArraySize</td>
</tr>
<tr>
<td>6</td>
<td>Read input / @Read_Line</td>
<td>60%</td>
<td>∞ ≤ x.Size&lt;br&gt;Input = unknown</td>
</tr>
<tr>
<td>7</td>
<td>Src[ ] := SLENGTH(Variable)</td>
<td>50%</td>
<td>Buff_Size + Min(Var_Str.Size, n.length) ≤ Buff_MaxSize - 1</td>
</tr>
<tr>
<td>8</td>
<td>If statement</td>
<td>70%</td>
<td>Dest_Max_buff == Src_Var_size&lt;br&gt;Dest_Value = Src_Value</td>
</tr>
<tr>
<td>9</td>
<td>SLENGTH(Dst) = SLENGTH(Src)</td>
<td>70%</td>
<td>Min(Src_VarSize, n.length) ≤ Dst_VarSize</td>
</tr>
</tbody>
</table>

The VCs weight was used as the iteration method with several experiments in order to set the percentage as shown in the Table 7-2 and 7-3 indicating the threshold for the detection points. SPSS has been used for the statistical analysis.

**Table 7-3 Weight of the Total Detections.**

<table>
<thead>
<tr>
<th>Total Weight</th>
<th>Total Affect Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOW</td>
<td>0% - 39%</td>
</tr>
<tr>
<td>MEDIUM</td>
<td>40% - 67%</td>
</tr>
<tr>
<td>HIGH</td>
<td>68% - 100%</td>
</tr>
</tbody>
</table>
Chapter 7: Experiments, Analysis and Evaluation

![Figure 7-1: Vulnerability Pattern Weight with Total Affect.](image)

The blue and red lines indicate to the table 7-3 which shows the weight range of total affect. However, the green line indicates to the table 7-2 Weight for Detecting Vulnerable Code. As any percentage between the blue and red lines shows the weight of the risk for the vulnerabilities.

### 7.3.1 Conditions for Detecting Vulnerability in WSL

The conditions for detecting vulnerability within WSL as modelled in the current work entails the following:

1. Detect Buffer Overflow for string comparison
2. Identify the variable which has the String copy
3. Check for out of Bounds
4. Check for Buffer Overflow (String Sub)
5. Check for Buffer Overflow (Negative Index)
6. Check for Buffer Overflow (String and Read Bugs)

Table 7-4 provides a list of the vulnerability patterns that exist in each example. To distinguish between vulnerability and non-vulnerability the variables were examined to ascertain if the same variable has been used with most of the potential vulnerability functions or not. The overall vulnerability detection system constructed in this work was based on functions summarised in Table 7-4.
Chapter 7: Experiments, Analysis and Evaluation

Table 7-4 Vulnerability Pattern in C with Equivalent Pattern in WSL.

<table>
<thead>
<tr>
<th>Function Number</th>
<th>Name</th>
<th>Functions in C</th>
<th>Functions in WSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Refers to string/integer</td>
<td>%s / %d</td>
<td>@String/@String_To_Num</td>
</tr>
<tr>
<td>F2</td>
<td>Memory allocation</td>
<td>malloc()</td>
<td>ARRAY()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>calloc()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>realloc()</td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>String manipulation</td>
<td>strcpy()</td>
<td>X := Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>strcat()</td>
<td>String copy or Copy Variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>strcmp()</td>
<td>SUBSTR</td>
</tr>
<tr>
<td>F4</td>
<td>Memory copying</td>
<td>memcpy()</td>
<td>X[] := Y[]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>memmove()</td>
<td>Out bounds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>memccpy()</td>
<td></td>
</tr>
<tr>
<td>F5</td>
<td>Array access</td>
<td>variable[n] := variable (out of bound)</td>
<td>Negative Index variable[n] := variable (out of bound)</td>
</tr>
<tr>
<td>F6</td>
<td>Format string Functions</td>
<td>printf()</td>
<td>PRINT()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sprintf()</td>
<td>PRINFUSH ()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fprintf()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>vsprintf()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scanf()</td>
<td></td>
</tr>
<tr>
<td>F7</td>
<td>Insecure by design</td>
<td>gets()</td>
<td>@Read_Line(Standard_Input_Port)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stdin/stdout</td>
<td>String and read bugs</td>
</tr>
<tr>
<td>F8</td>
<td>String size</td>
<td>Sizeof(variable)</td>
<td>SLENGTH(str)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strlen(str)</td>
<td>LENGTH(str)</td>
</tr>
</tbody>
</table>

The following equation was employed to calculate the final result of the outputs:

\[
TW = \frac{TLAP + TMAP + THAP}{TNF} \quad \text{(7-1)}
\]

Where: TW is the total weight of Final Decision, TLAP is Total Low Affect Percent, TMAP is total medium affect percent, THAP is total high affect percent, TNF is total number of functions.

7.3.2 Experiment 1: Non-Vulnerabilities Processes Experiment

(a) Aim of Experiment

The aim of the experiment is to check if the developed tool tables are able to distinguish between vulnerabilities and non-vulnerabilities and evaluate its overall efficiency.

(b) Procedure of Experiment

A collection of 30 non-vulnerability process samples was tested in this validation to show that these normal processes have not carried out the same steps of vulnerabilities behaviour which
lead to false positives. These processes are listed in Table 7-5, and were chosen because they are the active processes on Windows operating systems.

(c) Results of Experiment

The following table shows the Common Weakness Enumeration (CWE) that have been chosen to build or system to recognise the non-vulnerabilities and to test them to see the following result.

<table>
<thead>
<tr>
<th>Non-Vulnerability Program</th>
<th>Detected</th>
<th>Non-Vulnerability Program</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE – 1</td>
<td>No</td>
<td>CWE – 2</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 3</td>
<td>No</td>
<td>CWE – 4</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 5</td>
<td>No</td>
<td>CWE – 6</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 7</td>
<td>No</td>
<td>CWE – 8</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 9</td>
<td>No</td>
<td>CWE – 10</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 11</td>
<td>No</td>
<td>CWE – 12</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 13</td>
<td>No</td>
<td>CWE – 14</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 15</td>
<td>No</td>
<td>CWE – 16</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 17</td>
<td>No</td>
<td>CWE – 18</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 19</td>
<td>No</td>
<td>CWE – 20</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 21</td>
<td>No</td>
<td>CWE – 22</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 23</td>
<td>No</td>
<td>CWE – 24</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 25</td>
<td>No</td>
<td>CWE – 26</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 27</td>
<td>No</td>
<td>CWE – 28</td>
<td>No</td>
</tr>
<tr>
<td>CWE – 29</td>
<td>No</td>
<td>CWE – 30</td>
<td>No</td>
</tr>
</tbody>
</table>

In fact, Table 7-5 shows that none of these normal processes attempted to make buffer overflow and that none of the issued calls related to all of the categories with their rules. While some of these normal processes did issue calls from the categories, none did so from all of the categories and in the order that a program vulnerable would follow, despite attempts to make the processes demonstrate various forms of behaviour by considering as many typical user interactions with each normal process as possible. However, due to the fact that these normal processes have access to the program, they can be infected from outside. In other words, some vulnerability has the ability to disrupt normal processes and serve as an avenue to exploit vulnerabilities by terminating, overwriting or erasing a program.
7.3.3 Experiment 2: Vulnerabilities Program Experiment

(a) Aim of Experiment

The aim of the experiment is to check if the developed tool tables is able to distinguish between vulnerabilities and non-vulnerabilities, ascertain its efficacy and to see the extent to which false positive is decreased to reach a satisfactory level.

(b) Procedure of Experiment

30 vulnerabilities samples were collected from different vulnerabilities repositories (See Appendix D. The collection of vulnerabilities was chosen from various categories of system vulnerabilities. Therefore, as long as the vulnerability follows the steps of behaviour which have been previously explained, then the aim is that it would be detected by the approach presented in this work. There were two experiments carried out for vulnerabilities programs, namely, vulnerability analysis which was explained earlier in Chapter Four and vulnerability detection, the results of which will be explained in more detail in this chapter.

(c) Vulnerability Example Tested.

Table 7-6 Vulnerability Code in Assembly

<table>
<thead>
<tr>
<th>Assembly Example</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 _main:</td>
<td>[esp+8], eax</td>
</tr>
<tr>
<td>2 LFB12:</td>
<td>lea      eax,</td>
</tr>
<tr>
<td>3 push ebp</td>
<td>[esp+30]</td>
</tr>
<tr>
<td>4 mov ebp, esp</td>
<td>mov       DWORD</td>
</tr>
<tr>
<td>5 and esp, -16</td>
<td>PTR [esp+4], eax</td>
</tr>
<tr>
<td>6 sub esp, 48</td>
<td>lea      eax,</td>
</tr>
<tr>
<td>7 call ___main</td>
<td>mov       DWORD</td>
</tr>
<tr>
<td>8 mov eax,</td>
<td>PTR [esp], eax</td>
</tr>
<tr>
<td></td>
<td>mov       DWORD</td>
</tr>
<tr>
<td>DWORD PTR __imp___job</td>
<td>call _memcpy</td>
</tr>
<tr>
<td>9 mov DWORD</td>
<td>PTR [esp+40], 10</td>
</tr>
<tr>
<td>10 PTR [esp+8], eax</td>
<td>mov       DWORD</td>
</tr>
<tr>
<td>11 lea eax,</td>
<td>DWORD PTR [esp+40]</td>
</tr>
<tr>
<td>12 mov DWORD</td>
<td>mov       DWORD</td>
</tr>
<tr>
<td>PTR [esp], eax</td>
<td>PTR [esp+8], eax</td>
</tr>
<tr>
<td>13 call _fgets</td>
<td>lea      eax,</td>
</tr>
<tr>
<td>14 movzx eax, BYTE</td>
<td>[esp+30]</td>
</tr>
<tr>
<td>PTR [esp+30]</td>
<td>mov       DWORD</td>
</tr>
<tr>
<td>15 movsx eax, al</td>
<td>PTR [esp+4], eax</td>
</tr>
<tr>
<td>16 mov DWORD</td>
<td>lea      eax,</td>
</tr>
<tr>
<td>PTR [esp+44], eax</td>
<td>[esp+20]</td>
</tr>
<tr>
<td>17 mov eax,</td>
<td>mov       DWORD</td>
</tr>
<tr>
<td>DWORD PTR [esp+44]</td>
<td>PTR [esp], eax</td>
</tr>
<tr>
<td>18 mov DWORD</td>
<td>call _memcpy</td>
</tr>
<tr>
<td>PTR</td>
<td>leave     ret</td>
</tr>
<tr>
<td></td>
<td>34 LFE12:  nop</td>
</tr>
<tr>
<td></td>
<td>35</td>
</tr>
</tbody>
</table>

122
Chapter 7: Experiments, Analysis and Evaluation

The former experiment was carried out using existing tools to specify the steps of the behaviour of vulnerability. As shown in Table 7-7, these are the 30 vulnerability samples which were analysed to validate the theory.

Table 7-7: Test Vulnerabilities for Theory Validation

<table>
<thead>
<tr>
<th>Vulnerability Program</th>
<th>Detected</th>
<th>Vulnerability Program</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE - 1</td>
<td>Yes</td>
<td>CWE - 2</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 3</td>
<td>Yes</td>
<td>CWE - 4</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 5</td>
<td>Yes</td>
<td>CWE - 6</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 7</td>
<td>Yes</td>
<td>CWE - 8</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 9</td>
<td>Yes</td>
<td>CWE - 10</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 11</td>
<td>Yes</td>
<td>CWE - 12</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 13</td>
<td>Yes</td>
<td>CWE - 14</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 15</td>
<td>Yes</td>
<td>CWE - 16</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 17</td>
<td>Yes</td>
<td>CWE - 18</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 19</td>
<td>Yes</td>
<td>CWE - 20</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 21</td>
<td>Yes</td>
<td>CWE - 22</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 23</td>
<td>Yes</td>
<td>CWE - 24</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 25</td>
<td>Yes</td>
<td>CWE - 26</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 27</td>
<td>Yes</td>
<td>CWE - 28</td>
<td>Yes</td>
</tr>
<tr>
<td>CWE - 29</td>
<td>Yes</td>
<td>CWE - 30</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Another experiment was to show that these steps of vulnerability behaviour can be used to detect vulnerability program by using the prototype of this research as explained in Chapter 5. A number of chosen vulnerability from those analysed before have been observed and tested by the prototype. In addition, a collection of vulnerability which is unknown to the prototype has also been examined. The unknown ones have been chosen to test whether the prototype is able to detect previously unseen vulnerability which is one of the objectives of this research. Despite the fact that the prototype of this research can analyse one process at the same time, the vulnerabilities programs have been tested separately. Therefore, the vulnerabilities have been tested one by one and this has been done.

7.3.4 Experiment 3: Unknown Vulnerabilities Program Experiment

(a) Aim of Experiment

The aim of the experiment is to check if our tool can detect unknown vulnerabilities example to test the performance and the effectiveness.
(b) Procedure of Experiment

The following Three tables where it has been tested our tool that has been built to detect the vulnerability. An example of determining an intersection point of confluence (No. D. = Number of Detection, No. E. P. = Number of Examination Points, No. M. = Number of Misdetection (False negative), No. F. A. = Number of false alarm (False Positive), T. No. V. = Total Number of Vulnerabilities). The following table is the output of assembling to WSL.

Table 7-8: Evaluate Effectiveness of Vulnerability Detection with the ASM2WSL Output.

<table>
<thead>
<tr>
<th>Example Name</th>
<th>Vulnerability Analyzer</th>
<th>Memory analyzer</th>
<th>T. No. V.</th>
<th>Decision maker</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>V2</td>
<td>21</td>
<td>6</td>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>V3</td>
<td>31</td>
<td>25</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>V4</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>V5</td>
<td>19</td>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>V6</td>
<td>19</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V7</td>
<td>14</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>V8</td>
<td>11</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>V9</td>
<td>52</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>V10</td>
<td>18</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>V11</td>
<td>15</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>V12</td>
<td>15</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>V13</td>
<td>12</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>V14</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V15</td>
<td>16</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>V16</td>
<td>29</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>V17</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V18</td>
<td>23</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>V19</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V20</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
The following table is the output of C to WSL.

**Table 7-9: Evaluate Effectiveness of Vulnerability Detection with the C2WSL Output.**

<table>
<thead>
<tr>
<th>Example Name</th>
<th>Vulnerability Analyzer</th>
<th>Memory analyzer</th>
<th>T. No. V.</th>
<th>Decision maker</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>V4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>V5</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V6</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V7</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V8</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V9</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V10</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V11</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V12</td>
<td>15</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>V13</td>
<td>12</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>V14</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V15</td>
<td>16</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>V16</td>
<td>29</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>V17</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V18</td>
<td>23</td>
<td>6</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>V19</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>V20</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 7-10: Vulnerability Pattern Detection for 20 Examples.

<table>
<thead>
<tr>
<th>Programs Names</th>
<th>Vulnerability Exist</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>Vulnerability Detect</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Yes</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P2</td>
<td>Yes</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P3</td>
<td>Yes</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P4</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P5</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P6</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P7</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P8</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>Yes</td>
</tr>
<tr>
<td>P9</td>
<td>Yes</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P10</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P11</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P12</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P13</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P14</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>P15</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>P16</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>P17</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>P18</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>P19</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>P20</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 7-2: Vulnerability Pattern.

Table 7-8 and 7-9 shows that several vulnerabilities recognised by the prototype. It also shows that the prototype was able to detect both known and unknown system vulnerabilities. Vulnerability analysis tool was adopted to find the potential vulnerability code in job search task. The vulnerability analysis tool has got several parameters which influence the ultimate results of the software vulnerability detection, also the memory analyser will affect the final result.
Chapter 7: Experiments, Analysis and Evaluation

7.4 General Analysis of Obtained Overall Results

The results of the analysis of vulnerabilities for the both vulnerability analysis and memory analysis using existing tools, as explained in Chapter 5, are shown in Figure 7-3.

![Visualization of Results](image)

**Figure 7-3: Visualisation of the Results for Vulnerability Detection Analysis.**

The results listed above indicate that the majority of the vulnerabilities in the various classes were indeed detected in some existing systems. However, several vulnerabilities contradicted the theory of this research by not doing so. That is to say, the rules governing these vulnerabilities did not follow the multi-steps on which this research is based. The vulnerability might not occur if it is required to read input from outsourcing as the input might not conflict the policy of the vulnerability. Table 7-11 indicates the vulnerabilities that were verified.

**Table 7-11: Verification of the Result for all Tools Developed**

<table>
<thead>
<tr>
<th>Name</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation ASM2WSL</td>
<td>Verified</td>
</tr>
<tr>
<td>WSL Transformation</td>
<td>Verified</td>
</tr>
<tr>
<td>Memory Analyzer</td>
<td>Verified</td>
</tr>
<tr>
<td>Vulnerability Detection</td>
<td>Verified</td>
</tr>
<tr>
<td>Detection</td>
<td>Verified</td>
</tr>
</tbody>
</table>

7.5 Evaluation of the Developed Model and Classification of Vulnerability Reports

To evaluate the proposed model developed in this research, a set of experiments were performed on real-world vulnerabilities and benign executable processes. These experiments were outlined earlier and their results explained in detail in the present chapter. The evaluation of this approach is based on these results with the most significant one being whether the prototype
successfully detected known and unknown vulnerabilities. The criteria that are widely used to evaluate the performance of the detection algorithm is the Overall Correct Detection Rate (OCDR). The OCDR refers to the accuracy of the detection system in providing correct alerts to users. Thus, there is a percentage for incorrect alerts. Incorrect alerts are of two types:

(a) **False Alarms (FA):** refers to the number of incorrect alerts that the system triggered, despite the non-existence of error in the detecting points affecting their detection. A false positive is commonly known as a "false alarm", is a result that shows how a given condition has been satisfied but has not been really satisfied in actual sense.

(b) **Missed Detections (MD):** refers to a number of the errors in the detecting output not identified by the detection system. False negative, entails situations whereby test result shows that a condition failed, but is actually successful in the actual sense.

Therefore, the performance of a proposed detection algorithm can be measured with respect to False Alarm Rate (FAR), Missed Detection Rate (MDR), and OCDR. The effectiveness of the vulnerability detection approach developed in this work was tested on a number sample codes that contains buffer overflows and format string bugs. The number of vulnerabilities that were correctly identified were measured, including the number of false positives and false negatives. The use of tools based on static analysis for vulnerability detection could have been used whilst maintaining the integrity of the detection system. However, the less precise attributes of tools for static analysis renders this impossible. All vulnerability detection tools can exhibit both false positives and false negatives. A system is said to report a false negative in the event that vulnerability is not detected, indicating a failure in the detection scheme.

Due to this issue, it is generally accepted that a vulnerability detection scheme is more useful if it possesses the ability to detect more vulnerability compared to just reporting false negatives. Of equal importance is the number of false positives that is also reported given their tendency to intensify the efforts required for the verification of the result of a vulnerability detection analysis. Against this backdrop, the number of false negatives and false positives detection in the analysis provides a good indication of how effective a detection mechanism is. In the context of the current work, all these attributes were extensively demonstrated as detailed in the previous sections.

After the evaluation of the framework, it is important to compare the predicted values \( (V_p) \) with the actual values \( (V_a) \) in order to ascertain the correlation between the two. This can be achieved using measures of precision and recall, as illustrated in Figure 7-4:
Chapter 7: Experiments, Analysis and Evaluation

(a) The precision measures provide an indication of the number of components that were predicted to be vulnerable and actually turned out to be vulnerable. If the precision output is high, then it implies that the number of false positives is low. In the context of the current work, the efficiency of this predictor attribute is guaranteed.

(b) The recall is simply a measure of the number of components that were predicted to be vulnerable but turned out not to be vulnerable. If the number of recall is high, then it indicates that the number of false negatives is low. Again, in the context of the current work, this predictor proved to be effective.

<table>
<thead>
<tr>
<th>Actually has vulnerability reports</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted to have vulnerability report</td>
<td>Yes</td>
<td>True Positives (TP)</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>False Negatives (FN)</td>
</tr>
</tbody>
</table>

Recall

Figure 7-4: Recall and precision clarified. Recall is TP/(TP + FN); Precision is TP/(TP + FP).

Percentage of functions that are classified correctly based on the relation:

\[ \text{Accuracy} = \frac{TN + TP}{TN + FN + FP + TP} \]  
(7.2)

Probability of detecting vulnerability (recall) is calculated as:

\[ PD = \frac{TP}{TP + FN} \]  
(7.3)

Probability of misclassifying a good function is represented as:

\[ PF = \frac{FP}{TN + FP} \]  
(7.4)

It is known as a bad function (i.e. false alarm)

To compute how close is the result to the ideal point (pf, pd) = (0,1), the following expression is used:
Chapter 7: Experiments, Analysis and Evaluation

\[ \text{Balance} = 1 - \frac{\sqrt{(0 - PF)^2 + (1 - PD)^2}}{\sqrt{2}} \]  \hspace{1cm} (7-5)

Recall and precision measures are very good predictors for evaluating a vulnerability detection model. Recall provides the basis that a very good approach for evaluation should be able to identify all relevant software vulnerabilities of a system. Some of these vulnerabilities may not be detected due to the reshape of the syntactic forms and therefore the similarity with the original might not be recognised, although there is a relationship between the code fragments. The use of recall can assist in highlighting these key differences. Precision on the other hand indicates the level of high precision with which vulnerability is detected within software.

7.6 Discussion of Further Results Obtained from the Detection Tool Developed

WVAT, which is the vulnerability detection tool developed in this research was tested by running it on 80 samples whilst comparing it with the five sample programs with different tools as shown in the previous section. 5 out of the 20 examples are demonstrated in this section. To examine the effectiveness and efficiency of the techniques used in this work (i.e. the combination of taint analysis with slicing as well as value range with SSA analysis within a FermaT Transformation Engine), three sets of tests was carried out. In set one, the value range with SSA algorithm was disabled and analysis of the sample programs was performed based on taint analysis with slicing. In set two, taint analysis with slicing was disabled and analysis of the sample programs was performed based on value range analysis with SSA. Finally, the tests were repeated with both algorithms enabled. The developed tool was tested with 20 samples which involve vulnerabilities and non-vulnerabilities. It was observed that samples 1 to 14 contain vulnerabilities while samples 15 to 20 do not contain vulnerabilities.

When the model was operating based on taint analysis mode, it was able to detect all vulnerabilities in the sample programs, as indicated in Table 7-12. The vulnerability memory call in vulnerability one, three and twenty were the only false positive that was detected, which adjudged safe is given that untrusted data source were properly validated. However, the emergence of a false positive echoes the limitation of adopting taint analysis given its inability to detect vulnerabilities that occurs because of insufficient input validation. The results illustrated are based on taint analysis and slicing.
Table 7-12: Results of Running Vulnerability Analyser Based on Taint Analysis and Slicing.

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>25</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>7</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 6</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 7</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 8</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Vulnerability 9</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 10</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Vulnerability 11</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Vulnerability 12</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 13</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Vulnerability 14</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 15</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 16</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Vulnerability 17</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 18</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Vulnerability 19</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 20</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>75</strong></td>
<td><strong>21</strong></td>
<td><strong>54</strong></td>
</tr>
</tbody>
</table>

Running WVDT based on value range with SSA analyser for vulnerabilities detection, only 9 false positives and 18 false negatives were highlighted. The results are shown in table 7-13 Value range analyser has a competitive edge over taint analysis given its ability to identify data that are outside a valid range of a potential vulnerability. Despite this edge, a key disadvantage of value range algorithm is that it lacks the ability to identify untrusted data and forms the basis for its failure to detect vulnerability in the form of format string. The combination of both algorithms within a formal transformation technique as demonstrated in this research improves the overall accuracy and precision of the vulnerability detection mechanism. The results are indicated in Table 7-14. By integrating both analyses, WVDT was able to detect all the 20 vulnerabilities with less false positives and false negatives. The results highlight the advantages of integrating both value ranges within a FermaT transformation engine, rendering it an effective and efficient approach for software vulnerabilities detection. The integration with value range analysis ensures that the accuracy the vulnerability detection is improved whilst minimising the number of false positives.
Table 7-13: Results of Running WVDT in Value Range with SSA.

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>26</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 6</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 7</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 8</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 9</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 10</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 11</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 12</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 13</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 14</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 15</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 16</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 17</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 18</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 19</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 20</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>58</td>
<td>9</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 7-14: Results of Running WVDT Involve Taint Analysis with Slicing and Value Range with SSA Transformations.

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>51</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>11</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Vulnerability 6</td>
<td>10</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 7</td>
<td>7</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Vulnerability 8</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Vulnerability 9</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Vulnerability 10</td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Vulnerability 11</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Vulnerability 12</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Vulnerability 13</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Vulnerability 14</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Vulnerability 15</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 16</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Vulnerability 17</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Vulnerability 18</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vulnerability 19</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Vulnerability 20</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>133</td>
<td>30</td>
<td>72</td>
</tr>
</tbody>
</table>
Chapter 7: Experiments, Analysis and Evaluation

Figure 7-5: Result of WVDT.

The data analysis showed the total number of false positive and false negative varied from one track to another. Table 7-12 to 7-14 lists the misdetection (MDs) and false alarms reported in each data. In the first track for the method slicing with taint analysis, there were some false alerts and some MDs. While in the second track for the method SSA with value range, where the false alerts and some MDs were much greater. In the final track, the number of false alarms was 30, and the number of MDs was 72. Here 5 vulnerabilities were selected, analysed and compared with other tools to compare their performance in terms of effectiveness and the efficiency.

Table 7-15: Evaluation Result on Effectiveness.

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Source</th>
<th>Total#</th>
<th>Confirmed#</th>
<th>Suspicious#</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CWE-1)</td>
<td>3625.25</td>
<td>BOF</td>
<td>7</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(CWE-2)</td>
<td>3785.52</td>
<td>BOF</td>
<td>51</td>
<td>32</td>
<td>19</td>
</tr>
<tr>
<td>(CWE-3)</td>
<td>3895.45</td>
<td>BOF</td>
<td>7</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>(CWE-4)</td>
<td>7625.78</td>
<td>BOF</td>
<td>10</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>(CWE-5)</td>
<td>7625.78</td>
<td>BOF</td>
<td>11</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>
Chapter 7: Experiments, Analysis and Evaluation

Figure 7-6: Result of Effectiveness.

Table 7-16: Evaluation Result on Efficiency.

<table>
<thead>
<tr>
<th>Name</th>
<th>File Size</th>
<th>Detection Time</th>
<th>IR Size</th>
<th>Vulnerability analysis Time</th>
<th>Memory Analysis Time</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CWE-1)</td>
<td>27KB</td>
<td>3321s</td>
<td>5KB</td>
<td>1711s</td>
<td>1610s</td>
<td>Yes</td>
</tr>
<tr>
<td>(CWE-2)</td>
<td>68KB</td>
<td>4928s</td>
<td>12KB</td>
<td>2615s</td>
<td>2313s</td>
<td>Yes</td>
</tr>
<tr>
<td>(CWE-3)</td>
<td>29KB</td>
<td>3263s</td>
<td>6KB</td>
<td>1845s</td>
<td>1418s</td>
<td>Yes</td>
</tr>
<tr>
<td>(CWE-4)</td>
<td>50KB</td>
<td>3549s</td>
<td>9KB</td>
<td>1958s</td>
<td>1585s</td>
<td>Yes</td>
</tr>
<tr>
<td>(CWE-5)</td>
<td>49KB</td>
<td>3819s</td>
<td>9KB</td>
<td>1924s</td>
<td>1895s</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 7-7: Result of Efficiency.

The Accuracy for each data set was calculated in order to show the total rate of the developed tool. Table 7-17 shows the PD, PF, Balance and Accuracy for each data set. The Accuracy indicates the developed algorithm and provides a measure of its integrity.
Chapter 7: Experiments, Analysis and Evaluation

Table 7-17: Level of Accuracy of Each Data Set.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Levels Check (Detected)</th>
<th>Environment</th>
<th>PD(%)</th>
<th>PF(%)</th>
<th>Balance</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>7</td>
<td>Windows (Static)</td>
<td>3</td>
<td>1</td>
<td>0.04</td>
<td>75.29%</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>51</td>
<td>Windows (Static)</td>
<td>2</td>
<td>1</td>
<td>0.03</td>
<td>66.23%</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>9</td>
<td>Windows (Static)</td>
<td>4</td>
<td>2</td>
<td>0.05</td>
<td>66.23%</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>0</td>
<td>Windows (Static)</td>
<td>3</td>
<td>1</td>
<td>0.04</td>
<td>75.58%</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>19</td>
<td>Windows (Static)</td>
<td>4</td>
<td>2</td>
<td>0.05</td>
<td>66.23%</td>
</tr>
</tbody>
</table>

Figure 7-8: Result of Accuracy.

In Table 7-17, the second column shows the total number of points that passed the three integrity checks. With the integrity algorithm, the detection of fault within the dataset will be impossible. As such, a data that is adjudged faulty will be labelled MD. As shown in Table 7-17 (7, 51, 7, 10 and 11) points from data set 1, 2, 3, 4 and 5, respectively, are MDs, where fewer false alarms can be detected. The OCDR in Table 7-18 for each data set was found to be 73.33% for dataset1 89.78% for dataset2 78.23% for dataset3 78.23% for dataset4 45.43% and for dataset5 25.25%. This result is more accurate than the result obtained after applying the integrity algorithm. Hence, the proposed integrity algorithm can support vulnerability detection in low level language.

Table 7-18: Overall Correct Detection Rate (OCDR) with Integrity Method.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Recall (%)</th>
<th>Precision (%)</th>
<th>MDs</th>
<th>MDR</th>
<th>OCDR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>73.33%</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>2</td>
<td>0.5</td>
<td>2</td>
<td>4</td>
<td>89.78%</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>4</td>
<td>1.5</td>
<td>3</td>
<td>5</td>
<td>78.23%</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>45.43%</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>4</td>
<td>1.5</td>
<td>4</td>
<td>6</td>
<td>25.25%</td>
</tr>
</tbody>
</table>
Chapter 7: Experiments, Analysis and Evaluation

7.6.1 Test Vulnerability with FermaT Metrics

This section provides a description of the metrics associated with FermaT used in this work. The Raw WSL is equivalent to the original assembler as represented in WSL, and the structured WSL represents the same code after restructuring. The following metrics types as indicated in Table 7-9 and 7-20 will produce a report about the vulnerability to help end user to improve the quality of the code. WSL matrices can be used to analyse the vulnerability detection code to provide more details in term of the complexity of the code and calculate the branches and nodes with a possible range of value of each variable. Tables 7-9 and 7-20 shows the figures from WSL metrics which helped to detect the vulnerabilities.

Table 7-19: Matrices before the Slicing.

<table>
<thead>
<tr>
<th>Vulnerable</th>
<th>Statements</th>
<th>Expressions</th>
<th>McCabe</th>
<th>Essential</th>
<th>CFDF</th>
<th>Branch-Loop</th>
<th>Structural</th>
<th>Time-execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vuln1</td>
<td>18</td>
<td>66</td>
<td>3</td>
<td>1</td>
<td>27</td>
<td>1</td>
<td>215</td>
<td>1674s</td>
</tr>
<tr>
<td>Vuln2</td>
<td>16</td>
<td>72</td>
<td>1</td>
<td>1</td>
<td>30</td>
<td>0</td>
<td>234</td>
<td>0s</td>
</tr>
<tr>
<td>Vuln3</td>
<td>22</td>
<td>80</td>
<td>1</td>
<td>1</td>
<td>36</td>
<td>0</td>
<td>264</td>
<td>1s</td>
</tr>
<tr>
<td>Vuln4</td>
<td>41</td>
<td>205</td>
<td>3</td>
<td>1</td>
<td>89</td>
<td>1</td>
<td>624</td>
<td>1948s</td>
</tr>
<tr>
<td>Vuln5</td>
<td>41</td>
<td>207</td>
<td>3</td>
<td>1</td>
<td>87</td>
<td>1</td>
<td>632</td>
<td>2500s</td>
</tr>
</tbody>
</table>
Chapter 7: Experiments, Analysis and Evaluation

Table 7-20: Matrices after the Slicing.

<table>
<thead>
<tr>
<th>Vulnerable</th>
<th>Statements</th>
<th>Expressions</th>
<th>McCabe</th>
<th>Essential</th>
<th>CFDF</th>
<th>Branch-Loop</th>
<th>Structural</th>
<th>Time-execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vuln1</td>
<td>6</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>70</td>
<td>558s</td>
</tr>
<tr>
<td>Vuln2</td>
<td>5</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>66</td>
<td>0s</td>
</tr>
<tr>
<td>Vuln3</td>
<td>7</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>58</td>
<td>0s</td>
</tr>
<tr>
<td>Vuln4</td>
<td>13</td>
<td>72</td>
<td>3</td>
<td>0</td>
<td>29</td>
<td>0</td>
<td>42</td>
<td>649s</td>
</tr>
<tr>
<td>Vuln5</td>
<td>13</td>
<td>72</td>
<td>3</td>
<td>0</td>
<td>23</td>
<td>0</td>
<td>206</td>
<td>833s</td>
</tr>
</tbody>
</table>

In the next section, a comparison of result from the current tool and other tools is presented. This is important as it offers a reliable way to validate the tool developed in terms of the overall performance, efficiency, and accuracy.

7.7 Comparison of Output Results from this Work with Other Past Works

As highlighted in the preceding chapters of the current work, a number of tools have been proposed for vulnerability detection. In an attempt to ascertain the performance of the tool developed in this work, it is important to examine the outputs of other tools. This will provide an indication of the added value which the current work has added by adopting a completely new approach. In the subsection that follows, a brief description and output results from other tools is presented.

7.7.1 Cqual

This tool is programmer-dependent given that it is based on annotation types which are provided by the programmer. Compared to tools based on lexical analysis, its detection rate is relatively poor but it possesses the ability to establish a fine distinction between false positives and real vulnerabilities which is an important attribute. Sample result outputs based on vulnerability detected is provided in Table 7-21.

Table 7-21: Results of Running Cqual Program Detected False Positives False Negatives

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Chapter 7: Experiments, Analysis and Evaluation

7.7.2 Goanna

This tool recorded a very high vulnerability detection rate in comparison to other tools but performs lesser when specifically compared to those based on lexical analysis. This is because Goanna cannot detect vulnerabilities in situations where pointers are employed for taint analysis given its lack of appropriate pointer aliasing. Table 7-22 shows outputs results based on Goanna where it was able to detect false positives and false negatives vulnerabilities.

Table 7-22: Results of Running Goanna

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

7.7.3 SPLINT

SPLINT employs techniques from dataflow analysis for vulnerability detection purposes and has been found to be extremely effective in tracking vulnerabilities in systems developed using C programming language. It has the capability to detect programming problems including buffer overflow, memory leaks, unused variables etc. A distinguishing feature of SPLINT is that it possesses language extension which can be adopted to define new attributes within a program code to enhance vulnerability detection. Table 7-23 gives illustrative example of how SPLINT is able to detect false positives and false negative vulnerabilities.

Table 7-23: Results of Running SPLINT.

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

7.7.4 Vulncheck

This is a GCC compiler extension endowed with the capability to issue warnings when vulnerabilities are detected within a program and works on the principles of dataflow analysis for
tracking variables that are tainted. Vulncheck employs value range propagation to improve the accuracy of tests based on taint analysis thereby minimising the number of false positives. Table 7-24 shows sample results in terms of false positives and false negatives detected.

### Table 7-24: Results of Running Vulncheck with Taint Analysis and Value Range Propagation Mode.

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### 7.7.5 VSA-Based Approach

The value set analysis (VSA) tries to compute a value set, i.e. a set of potential values, for every register and every variable. The value sets can contain both integer and pointer values. The VSA is based on the observation that the possible effects of an instruction can be estimated if the sets of potential values for all used operands are known. Thus, buffer overflows cannot reliably be detected without information about the potential values of the variables or, in the case of assembler code, the operands. In order to retrieve this information, a technique called value set analysis, which computes the sets of potential values for all registers and all locations in the memory is used. Table 7-25 provides an indicative result in terms of false positives and false negative vulnerabilities detected.

### Table 7-25: Results of Running VSA-Based Approach

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

#### 7.7.6 BOON

BOON adopts size and usage to model buffer by converting a buffer overflow into an integer constraint. This is then used to detect vulnerabilities by producing constraints for C string
Chapter 7: Experiments, Analysis and Evaluation

manipulation. Table 7-26 provides sample results, highlighting the number of false positives and false negatives reported.

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

7.7.7 PolySpace

This is a commercial tool that leverages advanced formal methods such as interpretation of high level abstraction with the sole aim of detecting vulnerabilities including division by zero, buffer overflows, and other errors emanating from program source code. Table 7-27 provides sample results, highlighting the number of false positives and false negatives reported.

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>4</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

7.7.8 VulMiner

VulMiner is a vulnerability detection or prediction tool that leverages static analysis with machine learning combining prediction methods. Table 7-28 provides sample results, highlighting the number of false positives and false negatives reported.

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
Chapter 7: Experiments, Analysis and Evaluation

7.7.9 BitBlaze

BitBlaze focuses on building a unified binary analysis platform and using it to provide solutions to a broad spectrum of different security problems. The binary analysis platform is designed to enable accurate analysis, provide an extensible architecture as well as program verification techniques to satisfy the common needs of security applications. By extracting security-related properties from binary programs directly, BitBlaze enables a principled, root-cause based approach to computer security.

Table 7-29: Results of Running BitBlaze

<table>
<thead>
<tr>
<th>Program</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vulnerability 5</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>3</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7-30 below provides an overall summary of the false positives and false negatives based on the current work and past work. As indicated, the current work outperforms other tools demonstrating the advantages of integrating taint analysis with value range analysis within a WSL framework through optimised transformation engine. This is one of the hallmark of the current work, signalling its contribution to knowledge and practice. The overall comparison is schematically illustrated as shown in Figure 7-30.

Table 7-30: Summary of Comparison between the Current Work (WVDT) and Other Works

<table>
<thead>
<tr>
<th>C Code Tools</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulncheck</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Goanna</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>equal</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>SPLINT</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>BOON</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PolySpace</td>
<td>4</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Code Tools</th>
<th>Detected Vulnerabilities</th>
<th>False Positives</th>
<th>False Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>WVDT</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>VSA</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>VulMiner</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>BitBlaze</td>
<td>3</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
7.7.10 Detecting Known and Unknown Vulnerabilities

This section discusses the testing and determines whether or not the prototype developed and presented in this work has successfully detected both known and unknown vulnerabilities. As noted above, the previous software detects only previously seen vulnerabilities, because, in order to have a vulnerability pattern in the database, the vulnerability first needs to be analysed. Therefore, traditional software fails to detect unknown or formerly unseen vulnerability, leaving systems vulnerable. By contrast, despite a number of false negatives among the results reported in Section 7.3.2, the prototype model presented in this work has been shown to have the ability to detect both known and unknown vulnerabilities.

Thus, while existing traditional software succeeds in detecting systems vulnerabilities from known system vulnerabilities, it fails to deal with new ones, making it likely to produce a number of false negatives at any time, given new vulnerabilities which are said to appear every day and which will not be detected because they are unknown to the detection products. By contrast, the prototype developed in this work has the proven ability to detect not only known vulnerabilities but also unknown ones, as long as they behave the same as previously analysed ones, giving it advantage competitive edge over traditional vulnerabilities detection tools. In other words, some false negatives are produced by the detection technique presented in this work and other traditional detection software, but the latter fails totally to detect previously unseen

Figure 7-10: Results of Comparing Vulnerability Detection Analysis of Various Tools with the Current Work.
vulnerabilities, while the prototype developed in this work can detect them. The proposed approach also detects the known and unknown vulnerability.

7.7.11 Discussion

The experiment demonstrated in this chapter indicates how vulnerability detection can be achieved using a number of tools. In particular, it highlights the competitive edge that the new model developed in this work as compared to other works. To detect vulnerability in codes, several methods can be employed as already highlighted in subsequent chapters. Slicing technique, for instance has been adopted extensively for vulnerability detection functions. However, it has a major flaw in that it cannot identify variables that are affected within a vulnerable code. The use of single static assignment (SSA) technique has been adopted by some researchers to address this limitation but its detecting capability can be very slow, when identifying problems with large codes where accuracy and effectiveness is very important. Against this backdrop, the current work combines both techniques. By integrating the merits of both techniques, the fundamental limitations and errors associated to each method was largely eliminated, improving precision and accuracy of vulnerability detection mechanism. Furthermore, the fusion of both methods ensures that the system is more capable and powerful in terms of its overall performance in terms of computational speed and accuracy of detection.

The framework developed in this work is for the prediction and detection of vulnerabilities in low level language such as assembly language using Wide Spectrum Language (WSL) and its Transformation engine, with a specific focus on buffer overflow which causes memory leakage. WSL is very powerful and easy tool to use. Its ability to automatically convert codes from one language to another language makes it possible to adopt it to perform number of appropriate diagnosis. The use of WSL facilitates the process of understanding the language of the code, because it contains a combination of annotations and executive style, which facilitate the process of understanding and the implementation of the code and analysis without affecting the software system itself. However, WSL in its current format and structure which has some limitations when it comes to vulnerability detection of low level language. As such, the current work extends and optimizes the WSL framework by enhancing its vulnerability detection capabilities using program slicing combined with taint analysis and static single assessment combined with value range analysis. This was achieved through identification of vulnerability rules and their enforcement into the detection system. Combining these methods for the purpose of vulnerability detection purpose makes this work a truly unique one. The novel system developed in the current work has the ability to detect coding problems through binary executable files,
Chapter 7: Experiments, Analysis and Evaluation

without access to the source code, thereby allowing for easy detection of vulnerable codes at a faster pace.

The overall mechanism and methodical framework in this work constitute a detailed approach for vulnerability detection given that it was based on formal methods, underpinned mathematical models thereby enhancing the accuracy and precision with which vulnerability is detected. The transformations methods used in this research assisted in simplification of codes and vulnerabilities extraction, achieving improvements without altering the fundamentals codes. In this work, risk pertaining to vulnerability detection was classified based on a number of criteria namely precision and call time. Classification of risk helps to establish if there are any weaknesses in the code. This helps to improve the detection of risks in the future. The developed system was tested using more than 80 examples to evaluate the system carefully. The study was able to demonstrate that system has the ability to detect any error present in the codes. We also selected and tested 20 random samples, and the results were 92% correct.

7.7.12 Dataset Selection

A collection of 80 vulnerabilities samples, and 30 normal processes that are running in Windows 7, were analysed and tested in this research. Given that the approach taken in this research targets Windows 32-bit program vulnerabilities, those analysed and tested were selected from the Win32 category and from the two repositories explained in Chapter 5. The purpose of these repositories, as their websites suggest, is to provide a collection of program vulnerabilities in order to analyse them and protect operating systems from them. Therefore, the classification of these vulnerabilities is not very significant or accurate. In other words, a piece of vulnerability may be wrongly categorised as vulnerability when it is, in fact, some other type of vulnerability. Thus, some of the vulnerability analysed here may not have been program vulnerability and this supports the second justification of apparent false negatives returned by the prototype in this work, as explained in Section 7.3.2.

7.7.13 Vulnerability Detection

After examining the results of all the samples analysed and tested, two conclusions can be drawn about the theory of vulnerabilities. First, given that no false positives occurred when the benign processes were tested, it can be said that the theory of vulnerability is unique to vulnerable program and not to normal processes. It can, therefore, be concluded that the theory of vulnerability is a characteristic that distinguishes between the vulnerable program and normal
Chapter 7: Experiments, Analysis and Evaluation

processes. The second conclusion is that while the performance and usability of the operating system may be reduced while the detection is running, this is not an insurmountable obstacle, because the behaviour-based vulnerability detection can be improved to enhance performance and usability in order to meet the need of end-users to browse the system easily while detection occurs on the fly.

Regarding the false negatives produced by both the analysed and tested vulnerabilities, three observations can be made, each having its own unique solution. First, if candidate vulnerability can be considered to be some type of malware other than vulnerability, as the definition of vulnerability suggests. Secondly, vulnerability may be expected to do anything; for example, it may build its own kernel API driver and may issue other types of API and Native API calls, helping it to jump over commands to avoid detection. This would require a better understanding of the low level of the Windows operating system and better programming techniques, which are believed to be accomplishable. The third observation is that these vulnerabilities may occur while executing, but our prototype lacks some of the functionality and is not implementable. The best solution to this problem is to complement the approach used in this study with other known approaches. It is assumed that the level of true positives will be sustained or increased by the combination, while the number of false negatives will be reduced. WSL with FermaT can help to cover most of the points.

7.8 Summary

This chapter provides a detailed evaluation scheme that was used to verify and validate overall performance of the new model developed in terms of efficiency and effectiveness. Several factors, in particular, speed detection and accuracy was evaluated. The results obtained were promising compared to related work in this field. Several experiments were carried out illustrate how the overall model developed in this work was verified and validated. Three main experiments were carried out namely: vulnerabilities detection, normal processes and system performance and usability.
Chapter 8 Conclusion and Future Work

Objectives

- Provide a thesis summary of this research.
- Highlight the significance of contributions to knowledge.
- Describe the criteria upon which the current work can be adjudged successful.
- Highlight the limitations of the work.
- Present an agenda for future possible extension of the work.
8.1 Thesis Summary and Concluding Remarks

The goal of this chapter is to describe how the research aims and objectives listed in Chapter One are met. To reiterate, the central aim of this research is to investigate and develop adequate techniques for detecting and analysing vulnerability patterns in binary executable files of assembly language programming, using formal methods based on WSL and its associated program transformation techniques. Essentially, the focus of the work is to extend and deepen the knowledge of the field of vulnerability detection in binary executable files powered by low level programming language such as assembly language. Against this backdrop, the overall aim of the research could be said to have been achieved based on a number of research activities and numerous analyses that have been embarked upon, as highlighted in the succeeding sections.

Based upon the findings from the literature review detailed in chapter Two; the establishment of the statement of the problem which the current work seeks to address; the overall research design and methodological approach described in chapter Three; the final design of a robust algorithm and mechanism for Vulnerability Detection and implementation presented in chapters Four and Five, respectively; and the overall evaluation and validation of the tool discussed in chapter Six; the summary of conclusions, main findings and original contributions to knowledge stemming from this research is presented in this chapter. Also presented are the limitations of the research, as well as possible future extension of the current work.

In Chapter Two, a detailed review of extant literature was carried out with the view to identify the gap in knowledge which the current research seeks to fill. In particular, an overview of binary executable files in assembly language programming and their role in enhancing vulnerability threats in computer science and software engineering parlance was presented. The justification for focusing on binary executable files was also highlighted. More importantly, a review of key concepts and techniques in software engineering which will be utilised in order to fill the gap was carried out. This was achieved through the identification of the strengths and weaknesses of each technique which then informed the approach taken to realise the aim and objectives of the current work. A number of researchers have developed various techniques and models for detecting vulnerabilities in binary executable files of assembly language but most of the works found in literature were based on informal procedures that can greatly affect the accuracy of the vulnerability detection. The current work therefore addresses this fundamental gap in knowledge by adopting FermaT Transformation Engine within a Wide Spectrum Language (WSL) framework based on formal methods for vulnerabilities detection in binary executable files.
Vulnerability detection system requires transformation Engine and Intermediate Language in order to perform analysis in a better and improved manner. However, for binary executable files powered by low level programming language such as assembly language, the detection of vulnerability (e.g. buffer overflow) is an extremely difficult proposition. Given that binary executable files constitute the major loopholes through which hackers gain access into a computer systems platform, it therefore becomes of paramount importance to develop new techniques and approaches for the detection of vulnerabilities in such systems. The development of such tools using formal methods based on proven mathematical algorithm within WSL framework through its transformation engines is the hallmark of the current research. The current research therefore improves the vulnerability detection and analysis of binary executable files in low-level programming language through the development of a robust and reliable vulnerability detection mechanism within a WSL framework using powerful transformation techniques in a formal manner.

To achieve aim of the current work, based on limitations identified in literature, a robust systems architecture detailing the proposed vulnerability detection mechanism as part of the overall tool development was highlighted in Chapter Three. The architecture presented is able to detect the vulnerability in binary executable files using two techniques namely: (i) program slicing combined with static taint analysis and (ii) static single assignment augmented with value range analysis. The overall architecture of the model developed in this work is composed of three subsystems namely: translation, vulnerability analysis, and memory analysis, which are designed on the basis of Fermat Transformation techniques within a WSL framework. The vulnerability detection process which stems from the architecture constructed in Chapter three was developed into a practical tool as detailed in Chapter Four. The process takes into account 80 vulnerabilities that are related to buffer overflow, and the map matching process, using three phases of detection checks, which consist of: (i) vulnerability analyser, (ii) memory analyser, and (iii) decision maker. In the first phase, vulnerability analyser detecting process was used to identify the vulnerability functions of the program code. This is then followed by the use of memory analyser to check the value range of variable, adding a further layer of detection. In the final phase, decision maker system was employed, to ensure the validity and detection of the vulnerability.

The implementation strategy of the model development was illustrated extensively in Chapter Five with the performance of the proposed vulnerability detection process tested and evaluated as highlighted in Chapter Seven, using different code on vulnerability data collected from
different sources. In the testing process, more than 60 samples of data were used in order to validate the collected data. The evaluation result revealed that the proposed algorithm can accurately verify the detection of buffer overflow with 98.6% accuracy. The strong accuracy of the process emanated from layered detection checks which have the capacity to minimize the effect of errors that are related to vulnerability functions and variables.

Overall, this thesis proposed a novel vulnerability detection scheme for binary executable files powered by assembly language programming. Vulnerabilities were classified based on common features to identify potential threats in binary executable files. The approach adopted in this work was based on the combination of a number of techniques and concepts including taint analysis, program slicing, static single assignment value range process and FermaT Transformation Engine, all of which have been employed in one way or the other regarding the development of vulnerability detection model. The application of value range process to address the problem of vulnerability detection in low level language constitute one of key contributions of the current work to the field computer programming. The integration of the tool with a transformation engine can also have a significant positive impact on vulnerability detection mechanism in software parlance.

The experimental results highlighted in this thesis supports the notion that the integration of value range process and taint analysis within a transformation method offers a superior approach to vulnerability detection in binary executable files. The current tool which stems from this work generally outperform a number of existing tools as demonstrated on a variety of test cases. Specifically, it shows that as long as the need to translate from one programming language to another programming language will continue to be required, the need to develop the type of tools extensively demonstrated in this work cannot be overemphasized.

8.2 Major Contributions of the Current Research

Based on the overall presentation in this chapter, which demonstrates a model that utilises formal methods within a WSL framework for vulnerabilities detection system in binary executable files, the contribution of the current work to research, academics and industry practice can be summarised as follows:

(a) Development of a novel unified vulnerability detection framework and process for binary executable files powered by assembly language. The tool developed has the potential to ascertain the level of safety and reliability of the vulnerability detection tasks.
(b) Novel extension of the Wide Spectrum Language (WSL) based on formal methods through the incorporation transformation techniques for the detection of vulnerabilities binary executable files in assembly language programming. The overall tool was developed through the integration of two tools namely WSL and FermaT.

(c) Development of a robust architecture for vulnerabilities detection based on integration of a program transformations techniques such as program slicing in combination with taint analysis as well as single static assignment in combination with value range analysis. Such integration of multiple techniques within a unified modelling framework provides a better and improved vulnerability detection mechanism for binary executable files and minimises the rates of false negatives and false positives.

8.3 Success Criteria Revisited

The success criteria, which have been formulated in order to measure the success of this research, are provided in Chapter One. These criteria are revisited in this section in order examine whether they have been met or not. The criteria of success of the approach adopted in this work are emphasized based on how the set of research questions were answered and a number of other considerations as follows:

(a) What Type of Information must be Detected in order to Improve the Vulnerability Detection of Low Level Language through WSL?

The main components of vulnerability detection, include vulnerability analyser which checks the functions as to whether it is vulnerable based on pattern-matching algorithm; memory analyser which checks the value range of variables that are liable to certain errors and could provide misleading or faulty information to the user. Therefore, information derived from these components is highlighted for monitoring, with the aim of improving the detection of vulnerability in low level language through the WSL systems. This information includes variables data, and functions matching result. In addition, transformation data is considered as an essential factor for enhancing the pattern matching algorithm. Thus, detection of transformation was monitored to improve the detection process. A detailed explanation of this information is provided in Chapter Four.
(b) How can an Efficient Vulnerability Detection System for Low Level Language be Designed using the Concept of Transformation Engine?

The design of a vulnerability detection system within a WSL framework is presented in Chapter Three. The design is comprised of three main subsystems: program slicing, Static Single Assignment (SSA), and an application subsystem. These subsystems correspond to the main phases of the transformation engine system and formal method. The abstract layer of transformation framework is used to construct the components of the proposed system and it has been validated to be working appropriately.

(c) How can the Vulnerability Detection Technique be Designed Efficiently to Determine the both certain and Uncertain Vulnerability?

A novel technique for detecting the vulnerability of binary executable files within a WSL framework is developed and presented in Chapter Five. The proposed detection technique has the ability to detect any inconsistency related to functions, variable data, and pattern matching process, using three phases of detection checks. These phases are: (i) vulnerability analyzer, (ii) memory analyser, and (iii) decision maker. A decision maker system is applied in the final phase to identify any uncertainty related to the pattern matching process; a detailed explanation of the proposed algorithm is provided in Chapter Four. A thorough investigation illustrating how the proposed vulnerability detection technique is different from other existing vulnerability detection technique is presented.

The components of the proposed system are constructed using the layered FermaT transformation engine within a WSL framework. An investigation illustrating how the use of decision maker logic can positively affect the actual implementation of the process is presented. Finally, an investigation illustrating whether the detection technique can be implemented in the real world for commercial purposes was carried out. The proposed detection technique was implemented and tested using data collected in a real environment. The results suggest that the detection technique has the capability to support real world application, and thus can be used commercially.

(d) To What Extent are Vulnerabilities Detected in the Model Developed

This is one of the most significant criterions in this research. This criterion will measure the success of the proposed approach. Due to the fact that one of the reasons why behaviour-based
Chapter 8: Conclusion and Future Work

Vulnerabilities detection techniques were introduced is to detect vulnerabilities, the approach in this research must be able to perform this. The prototype’s results detailed in Chapter Seven demonstrate that the prototype developed is able to detect vulnerabilities through WSL. This conclusion is drawn due to the 10 specimens (including known and anonymous vulnerabilities) examined by the prototype in this work. However, there were a number of false negatives in the case of both analysed and tested vulnerabilities. The reasons for these false negatives were discussed in detail in Chapter Seven.

(c) False Positive Production.

Section 7.4.2 provides the results for the benign processes which were tested by our prototype. As illustrated in Table 7-5, none of these normal processes shows the possible way to give chance making the buffer overflow. As a result, it can be said that no false positives were produced by our prototype. This result can lead to a conclusion in which the characteristic of vulnerability is unique to Program vulnerabilities and can be used to distinguish between viral and normal processes. The adaptation of the main vulnerability detection technique is successfully extended to Process the relationship of transformation methods with traditional detection which helps in software engineering context.

8.4 Research Limitations

Although a number of novelties and innovations were demonstrated in the current work, the overall research is still tainted with some limitations as highlighted here. Most of the limitations are practical issues that stems from the implementation of this research or the nature of program vulnerabilities. As discussed in Chapter Seven, the results of the proposed technique are more positive than the results of many existing detection techniques. Despite the ability of the proposed technique presented in this thesis to be able to provide valid integrity alerts at a rate of roughly 99%, there exist some limitations as stated below:

(a) Testing environment: the performance evaluation of the proposed detection technique is based on a small data set, collected from different sources. However, it is vital to validate the effectiveness of the technique in more complex environments such as the banking sector or airline industry where large pool of dataset exist, in order to dully ascertain the usability and robustness of the technique. Therefore, further investigations are required to test the performance of the system in such environments.
Chapter 8: Conclusion and Future Work

(b) System continuity: the detection can experience outage while carrying out analysis on some code, for example, buffer overflow. As a result, the system continuity will be affected. Many new detection systems use simple pattern matching to augment the vulnerability and predict vulnerability functions. However, the proposed detection technique uses only few transformations method as the primary source of positioning data. Further research must be undertaken to address this issue with the ability to detect more vulnerability and malicious codes.

(c) The vulnerabilities analyser can detect the buffer overflow and need to be add any new vulnerabilities to detect through the WSL and this consider one of the limitation.

(d) The new vulnerability might not be detected unless modelling and adding the behaviour of it in the WSL extension tool.

(e) The current detection mechanism developed in this work cannot handle vulnerabilities that pertains to web applications.

8.5 Future Work

Despite the novelties and innovations demonstrated in the current work, further research work is required to reinforce the accuracy of the vulnerability detection scheme within the overall tool development. Accordingly, the research tasks that could form the basis of future work are discussed as follows. The inherent inability to analyse source codes that contains structures are one of the challenges to address when it comes to vulnerability detection and it can constitute a major impediment for practical applications of the concepts presented in this thesis. This pertains to the fact that structures are used in abundant quantities in assembly language programming because the infrastructure of dataflow within a GCC compiler cannot distinguish between assignments in different fields within a structure. The need to develop vulnerability detection mechanism that will address this issue is therefore pertinent.

One very common vulnerability that the implementation strategy in this work successfully detects is the classic string copy buffer overflow which most of the vulnerability tool fails to detect. The taint analysis and value range techniques adopted in this research performs well with integer and string variables, but there is a need for their extension to cover variants of vulnerability. Achieving this as part of a future work will improve the efficiency of such vulnerability detection systems. Many of the vulnerabilities that were discovered emanated through interactions between several functions in a program. WSL FermaT Transformation engine combined with inter-procedural and whole-program analysis makes the analysis possible.
Chapter 8: Conclusion and Future Work

and easier. The most challenging research for the future pertains to analysing the malicious and design security pattern and the evolution model in order to optimize the vulnerability code from insecure to more secure. The vulnerability detection system presented is very limited in terms of infrastructure for inter-procedural analysis and its ability to whole program optimization is also limited. Further research in this area is therefore required.

The literature review in this thesis demonstrates that the area of program vulnerabilities detection for the low level language is a challenging area due to the fact that it is a never-ending fight between attackers and defenders. Behaviour-based vulnerabilities detection techniques are especially interesting because they overcome the problems associated with traditional static vulnerabilities detection. As suggested by earlier, the nature of program vulnerabilities leads vulnerabilities researchers to seek alternative solutions. Therefore, future research for any vulnerabilities detection approach is needed in order to enhance it. Furthermore, the following points are some proposals for future research:

(a) Analysing other types of vulnerabilities to examine how they act within the translation and transformation. This task is likely to require an entire research team to realise given that it involves an enormous level of analysis of many types of vulnerabilities to determine their behaviour.

(b) As explained in Chapter 2, the problems associated with the heuristic-based vulnerabilities detection can be solved by using the behaviour-based vulnerabilities detection. Therefore, one the future research will be to investigate the use of the current research findings to solve these problems.

(c) Changing the WSL vulnerabilities code to other format to detect the vulnerabilities easily with different form will also offer an interesting research dimension.

(d) Transform the isolation code from vulnerable code or insecure code to secure code

(e) Guided/directed transformations for building secure systems from insecure components (to transform the code correctly with taking on the consideration of security part).
List of References


KINDER, J. (2010) Static analysis of x86 executables (Doctoral dissertation, Technische Universität Darmstadt)


Appendix A Comparison Vulnerability Detection Tools

Summary of the analysis based on buffer overflow (BOF) vulnerability detection works of some researchers which was based on the work by Shahriar et al. (2010).

Table A-1 Summary of the Analysis Based on Vulnerability Detection Work of Researchers.

<table>
<thead>
<tr>
<th>Work</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viega et al. (2002)</td>
<td>Tokenize source code and identify known vulnerable library function calls.</td>
</tr>
<tr>
<td>Evans et al. (2002)</td>
<td>Annotate a function body with pre- and post-conditions and warn users if conditions are violated.</td>
</tr>
<tr>
<td>Dor et al. (2003)</td>
<td>Detect all string manipulation errors through pointer analysis and contract checking on a subset of C language.</td>
</tr>
<tr>
<td>Ganapathy et al. (2003)</td>
<td>Detect BOF by forming integer range constraints and apply linear programming to solve these constraints.</td>
</tr>
<tr>
<td>Xie et al. (2003)</td>
<td>Detect BOF at each array and pointers dereference by applying path sensitive and inter-procedural symbolic execution.</td>
</tr>
<tr>
<td>Vulncheck (2005)</td>
<td>Apply tainted data flow analysis to detect BOF vulnerabilities.</td>
</tr>
<tr>
<td>Tevis et al. (2006)</td>
<td>Detect vulnerable library function calls in the section table of portable executable files.</td>
</tr>
<tr>
<td>Hackett et al. (2006)</td>
<td>Apply annotation language to specify buffer interfaces for safe usage of buffers in functions.</td>
</tr>
<tr>
<td>Le et al. (2008)</td>
<td>Apply query-based demand driven analysis to warn BOF and prioritize warnings based on path sensitivity.</td>
</tr>
</tbody>
</table>
Appendix A: Comparison Vulnerability Detection Tools

Classification of the analysis based on buffer overflow (BOF) vulnerability detection.

Table A-2 Classification of the Analysis Based on Vulnerability Detection.

<table>
<thead>
<tr>
<th>Work</th>
<th>Inference technique</th>
<th>Analysis sensitivity</th>
<th>Analysis granularity</th>
<th>Soundness</th>
<th>Completeness</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wagner et al. (2000)</td>
<td>Constraint</td>
<td>None</td>
<td>Statement</td>
<td>Yes (limited language feature)</td>
<td>No (absence of flow sensitive analysis)</td>
<td>C</td>
</tr>
<tr>
<td>Weber et al. (2001)</td>
<td>Constraint</td>
<td>Control flow, context</td>
<td>Control flow, system dependence graph</td>
<td>Yes (absence of analysis sensitivity)</td>
<td>No (absence of points to sensitive analysis)</td>
<td>C</td>
</tr>
<tr>
<td>Viega et al. (2002)</td>
<td>String pattern matching</td>
<td>None</td>
<td>Token</td>
<td>Yes (limited scope of the problem)</td>
<td>No (absence of semantic analysis)</td>
<td>C, C++</td>
</tr>
<tr>
<td>Evans et al. (2002)</td>
<td>Annotation</td>
<td>Control flow</td>
<td>Statement</td>
<td>Yes (limited scope of the problem)</td>
<td>No (absence of analysis sensitivity, assumption on code)</td>
<td>C</td>
</tr>
<tr>
<td>Dor et al. (2003)</td>
<td>Annotation</td>
<td>Points-to analysis</td>
<td>Intra-procedural</td>
<td>Yes (limited language feature)</td>
<td>No (absence of analysis sensitivity)</td>
<td>C</td>
</tr>
<tr>
<td>Ganapathy et al. (2003)</td>
<td>Constraint</td>
<td>Context, points-to analysis</td>
<td>System dependence graph</td>
<td>Yes (limitation of analysis sensitivity)</td>
<td>No (absence of flow sensitive analysis)</td>
<td>C</td>
</tr>
<tr>
<td>Xie et al. (2003)</td>
<td>Constraint</td>
<td>Path, context</td>
<td>Intra and inter-procedural</td>
<td>Yes (limited scope of the problem)</td>
<td>No (result interpretation)</td>
<td>C</td>
</tr>
<tr>
<td>Vulncheck (2005)</td>
<td>Tainted data flow</td>
<td>Value range</td>
<td>Statement, data flow, inter-procedural</td>
<td>Yes (limited scope of the problem)</td>
<td>Yes</td>
<td>C</td>
</tr>
<tr>
<td>Tevis et al. (2006)</td>
<td>String pattern matching</td>
<td>N/A</td>
<td>Statement</td>
<td>Yes (limited scope of the problem)</td>
<td>No (absence of semantic analysis)</td>
<td>x86</td>
</tr>
<tr>
<td>Hackett et al. (2006)</td>
<td>Annotation</td>
<td>N/A</td>
<td>Inter-procedural</td>
<td>Yes (limited scope of the problem)</td>
<td>No (assumption on code)</td>
<td>C</td>
</tr>
<tr>
<td>Le et al. (2008)</td>
<td>Constraint</td>
<td>Path and context</td>
<td>Inter-procedural</td>
<td>Yes (absence of analysis sensitivity)</td>
<td>No (result interpretation)</td>
<td>C</td>
</tr>
</tbody>
</table>
Appendix A: Comparison Vulnerability Detection Tools

Performance comparison of the analysis based on buffer overflow (BOF) vulnerability detection.

**Table A-3 Performance Comparison of the Analysis Based on Vulnerability Detection.**

<table>
<thead>
<tr>
<th>Inference</th>
<th>Work</th>
<th>FP</th>
<th>LOC</th>
<th>Manual effort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tainted data flow</strong></td>
<td><strong>Vulncheck (2005)</strong></td>
<td>0%</td>
<td>16</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Constraint</strong></td>
<td><strong>Wagner et al. (2000)</strong></td>
<td>90%</td>
<td>32K</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td><strong>Weber et al. (2001)</strong></td>
<td>N/A</td>
<td>18K</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td><strong>Ganapathy et al. (2003)</strong></td>
<td>N/A</td>
<td>68K</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td><strong>Xie et al. (2003)</strong></td>
<td>35%</td>
<td>1.6M</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td><strong>Le et al. (2008)</strong></td>
<td>N/A</td>
<td>93.5K</td>
<td>No</td>
</tr>
<tr>
<td><strong>Annotation</strong></td>
<td><strong>Evans et al. (2002)</strong></td>
<td>75%</td>
<td>20K</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td><strong>Dor et al. (2003)</strong></td>
<td>50%</td>
<td>228</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td><strong>Hackett et al. (2006)</strong></td>
<td>10%</td>
<td>400K</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>String pattern matching</strong></td>
<td><strong>Viega et al. (2002)</strong></td>
<td>57%</td>
<td>68K</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td><strong>Flawfinder (2004)</strong></td>
<td>50%</td>
<td>16K</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td><strong>Tevis et al. (2006)</strong></td>
<td>N/A</td>
<td>9.3M</td>
<td>No</td>
</tr>
</tbody>
</table>
Appendix B Define Other Components Related to System Architecture

(a) Intermediate Representation

An Intermediate representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive for further processing, such as optimization and translation. A good IR must be accurate – capable of representing the source code without loss of information – and independent of any particular source or target language. Using an intermediate language for this fall in complexity between high-level languages and low-level languages, such as assembly languages.

(i) Wide Spectrum Language

In the model framework, vulnerabilities are identified based on WSL by looking for general vulnerabilities that exist for all application on any languages and action system. The core of the FermaT transformation system is the WSL language based Morgan’s specification statement and Dijkstra’s guarded commands. The overall aim is to establish a language which acts as an intermediate language when processing a legacy system. WSL was designed for reengineering tasks and covers:

(i) Simple, regular and formally defined semantics
(ii) Simple, clear and unambiguous syntax
(iii) A wide range of transformations with simple, mechanically-checkable correctness conditions
(iv) The ability to express low-level programs and high-level abstract specifications

The heart of the WSL language is a very small and mathematically tractable kernel language. This language supports already all necessary operations needed for a programming and specification language. In the context of this tiny kernel language, it is relatively easy to prove the correctness of a transformation, but the language is not very expressive for programming. For that reason, the language is extended into an expressive programming language by defining new constructs in terms of the kernel.
Appendix B: Define Other Components Related to System Architecture

This extension is carried out in a series of layers with each layer building on the previous language level. There are several implementations of subsets of WSL in order to be executed which have different properties such as:

- **Interpreter:** The SCM interpreter (The Scheme interpreter used to execute the program and an integer is an atomic 32 bit value). Also, it does very little testing of parameter types, given that it is optimized for speed.

- **Compiler:** it has four compiler that helps in executing and analysing based on the following techniques:
  1) The Hobbit Scheme compiler (Compile the Scheme into C) which does very little testing of parameter types, since it is uniquely optimized for speed.
  2) The Gambit interpreter and compiler (gives an immediate error)
  3) The Bigloo interpreter and compiler (implementation of Scheme and large numbers will be implemented on the heap.)
  4) Guile is an interpreter and compiler for the Scheme programming language.

Note that any implementation of WSL will be "incomplete" (in the sense that there will be WSL programs which cannot be executed) because WSL includes infinitary logic and specification statements.

In WSL, local variables are implemented via dynamic binding, instead of static binding. WSL is translated to Scheme for interpreting or compiling and Scheme uses static binding. Dynamic binding is emulated in Scheme as follows: On entry to a VAR block the global value of a local variable is stored in a Scheme static bound local variable. Then the global variable is assigned the new value. On exit of the VAR block the global variable’s value is restored from the local variable. For example, this WSL program:

**Table B- 1 WSL Code Program**

<table>
<thead>
<tr>
<th>WSL Program</th>
<th>produces this output</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>Found new global: /foo</td>
</tr>
<tr>
<td>foo := 3;</td>
<td>Starting Execution...</td>
</tr>
<tr>
<td>PRINT(&quot;Initial global foo = &quot;, foo);</td>
<td>Initial global foo = 3</td>
</tr>
<tr>
<td>VAR &lt; foo := 4 &gt;;</td>
<td>Local foo = 4</td>
</tr>
<tr>
<td>PRINT(&quot;Local foo = &quot;, foo);</td>
<td>Within bar, foo = 5</td>
</tr>
<tr>
<td>bar();</td>
<td>After bar, still within VAR, foo = 5</td>
</tr>
<tr>
<td>PRINT(&quot;After bar, still within VAR, foo = &quot;, foo);</td>
<td>Outside VAR foo = 3</td>
</tr>
<tr>
<td>ENDVAR;</td>
<td>Execution time: 0</td>
</tr>
<tr>
<td>PRINT(&quot;Outside VAR foo = &quot;, foo)</td>
<td></td>
</tr>
<tr>
<td>WHERE</td>
<td></td>
</tr>
<tr>
<td>PROC bar() ==</td>
<td></td>
</tr>
<tr>
<td>foo := foo + 1;</td>
<td></td>
</tr>
<tr>
<td>PRINT(&quot;Within bar, foo = &quot;, foo) END</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B: Define Other Components Related to System Architecture

Table B- 2 Result of the Compiler in Scheme Language

It is translated to Scheme as follows:

```scheme
;; Scheme translation of WSL code
(letrec ((/bar (lambda ()
    (set! /foo (+ /foo 1))
    (display-list "Within bar, foo = " /foo)
    #t))

)(begin
    (set! /foo 3)
    (display-list "Initial global foo = " /foo)
    (let ((/foo-save /foo))
        (set! /foo 4)
        (display-list "Local foo = " /foo)
        (/bar)
        (display-list "After bar, still within VAR, foo = " /foo)
        (set! /foo /foo-save))
    (display-list "Outside VAR foo = " /foo)))
```

Note that all references to foo (/foo in Scheme) are to the global variable, the local variable /foo-save is used to save and restore the value of /foo. These files implement the SCM interpreter and runtime library which are used to execute WSL code translated to Scheme. The WSL code for FermaT is translated to Scheme, then compiled to C using the Hobbit Scheme compiler. The C code produced by the compiler uses the SCM runtime library. Translating WSL to binary is basically standard compiler technology.

When code written in a language is interpreted, its syntax is read and then executed directly, with no compilation stage. A program called an interpreter reads each program statement, following the program flow, then decides what to do and does it. A hybrid of an interpreter and a compiler will compile the statement into machine code and execute that; the machine code is then discarded, to be interpreted anew if the line is executed again.

- **Set of Regression tests/analysis**: is a statistical process for estimating the relationships among variables. It includes many techniques for modeling and analyzing several variables, when the focus is on the relationship between a dependent variable and one or more independent variables (or 'predictors'). More specifically, regression analysis helps one understand how the typical value of the dependent variable (or 'criterion variable') changes when any one of the independent variables is varied, while the other independent variables are held fixed.
Appendix B: Define Other Components Related to System Architecture

(b) Core System

Involves multi tools and the only large system implemented in WSL is the FermaT transformation engine. Also, it involves other tools such as Meta-WSL (Meta-WSL extensions to WSL for manipulating syntax trees and writing program transformations and involves a translator). The following are the primary of the FermaT System which is consider the largest system implemented in WSL.

(i) FermaT Transformation System

The FermaT Transformation System is a powerful industrial-strength program transformation system based on the WSL language. FermaT has been used successfully in several major assembler to C and assembler to COBOL migration projects involving the conversion of millions of lines of hand-written assembler code to efficient and maintainable C or COBOL code. The FermaT Transformation System can help organizations transform their legacy Assembler systems and applications to better support the current and future needs of the business. The FermaT system has evolved from SML’s experience in the development and implementation of its FermaT tools for Assembler documentation, transformation and migration. The FermaT Workbench and FermaT Migration Engine have been developed specifically to support Assembler code documentation, transformation and migration. Current versions of the tool provide developers support for Assembler code documentation, logic, data analysis, business rule identification and code migration from Assembler to C and COBOL.

(ii) FermaT Migration Engine Overview

The objective of the FermaT Migration Engine is to enable the migration of large, highly complex legacy systems from Assembler to higher-level language such as C or COBOL. Once migrated, these systems are substantially easier to maintain and can evolve faster to meet the changing needs of the business they support. Because of FermaT’s use of a unique, formally defined high-level language (WSL) and its specifically designed code transformations, the migration process can be automated. As a result, large legacy systems can be migrated quickly, requiring a fraction of the resources necessary to migrate the code manually.

The WSL FermaT transformation system is built on the transformation theory that has the following attributes.
Appendix B: Define Other Components Related to System Architecture

(i) Improving the maintainability (in particular, flexibility and reliability, and hence extending the lifetime) of existing mission-critical software systems;

(ii) Translating programs to modern programming languages;

(iii) Developing and maintaining safety-critical applications;

(iv) Extracting reusable components from current systems, deriving their specifications, and storing the specification, implementation, and development strategy in a repository for subsequent reuse;

(v) Reverse engineering from existing systems to high-level specifications, followed by subsequent reengineering and evolutionary development;

Core System: The Core System provides essential functionalities and consists of the toolsets and these toolkits will help a lot in terms of analysis and getting results also for providing and testing some solutions. The main toolset are:

Extension of Transformation Engine: Transformation Engine, which supports the automatic transformation of programs and models while keeping certain properties invariant, is at the heart place of the whole toolset. Transformation Engine transforms the system by using transformation rules. Transformation depends on matching detection. If inputs are matched with predefined pattern, the system will be rewritten according to the transformation rules. Transformation rules are actually a kind of knowledge, which can be reused. It is the only large system implemented in WSL and converts WSL program before execution into Scheme. Transformation Engine which provides the program and model transformation functionalities, identify transformation rules to help check thread and build rules/policies to identify these vulnerabilities based on the transformation engine with the proposed approach in vulnerabilities detection to derive more powerful vulnerability detection approaches. After translation, a set of rules is constructed to help check for a possible buffer overflow on a string or array which checks the array boundaries and terminates the program with an error if the boundaries are exceeded. This program is semantically equivalent to the original, for initial states in which no buffer overflow occurs, but changes the semantics on buffer overflow to something which is easier to detect. In Fermat, given a program \( P \), it can produce equivalent program \( P' \) that is more structurally efficient, etc. Such that: \( P = P' \).

The existing transformations developed in FermaT are based on the WSL language levels rising from the kernel language to the procedure/function level. Most of the transformations work are based on the basic constructs of WSL, such as WHILE, IF, VAR and so on. Those transformations which have been proved formally alter the syntax of WSL but preserve the
Appendix B: Define Other Components Related to System Architecture

semantics of the program. They have been applied in practical projects as well as academic experiments and proved the efficiency. In subsequent sections, the transformation extension based the extended WSL which has the value range propagation features or the domain features will be discussed. The transformations called boundary checking transformations in the proposed transformation bank. They are not the composite of the basic transformations, but the ones based on the WSL language extension. Referring the boundary checking transformations adopted are movement transformations, encapsulation transformation and wrapper transformation. The boundary checking transformations are proposed based on the extended vulnerability detection constructs in WSL.

**FermaT Translation:** FermaT translation is used for translating programs into modem programming languages. It often translates program written in obsolete assembler language to more modem languages such as C.
Appendix C Additional Information on Process of Vulnerability Detection

Part 1

The patterns under investigation possess the following characteristics that technically inclined user may exploit stack-based buffer overflows to manipulate the program to their advantage:

(a) Function does not check for buffer length such as reading input which relies on external data to control its behaviour function to read an arbitrary amount of data into a stack buffer. By overwriting a local variable that is near the buffer in memory on the stack to change the behaviour of the program - which may benefit the attacker.

(b) Function does not check buffer lengths and may very well overwrite memory zone contiguous to the intended destination whether copying on variables or comparison with the buffer overflow size which depends upon properties of the data that are enforced outside of the immediate scope of the code. Also relies on user input to control its behaviour, but it adds a level of indirection with the use of the bounded memory copy function this function accepts a destination buffer, a source buffer, and the number of bytes to copy. For example, Access controls (instruction processing): Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program’s implicit security policy.

(c) Functions does not check the buffer boundaries such as printing and saving the buffer overflow with incorrect value in term of size and type or different format. The code depends on properties of the data that are not verified locally.

(d) Putting the program into an infinite loop and by overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address as specified by the attacker, usually a user-input filled buffer.

(e) By overwriting a function pointer or exception handler, which is subsequently executed

(f) By overwriting a parameter of a different stack frame or a non-local address pointed to in the current stack context.
Appendix C: Additional Information on Process of Vulnerability Detection

The list of untrusted sources is application and system specific. Buffer overflows, generally lead to crashes and the taint analysis algorithm developed in this research is able to detect the potential vulnerability while translating in order to help in the investigation of vulnerability and in the developing phase of the vulnerability detection in WSL. The algorithm works with content mapping with patterns defined in the file then it translates to WSL as predefined properties for identifying the potential vulnerability. In our research we have identified a limited number of potentially vulnerable functions and language constructs.

(a) C2WSL

This routine translates and models C code into WSL code in such a way that certain classes of vulnerability translate to certain semantic behaviours in the generated WSL code, playing a significant role in terms of enhancing vulnerability detection in WSL. The overall objective of this approach is to assist in clarifying the vulnerabilities detected in C in a better way which is easier to detect and refine within WSL. This entails developing C code and modelling it within a WSL environment in order to highlight the behaviour of the C code. The focus is on call buffer overflow vulnerability and to model it in WSL by defining a single array or sequence which models the memory of the C program. Within the model, a C variable, string or array is represented by a subsequence in the memory model. The current tool development is based on the following steps:

(a) Existing vulnerability code was taken in one or more existing languages (e.g. C code).

(b) C code was analysed to determine the vulnerabilities on the code by tagging and marking the classes which have potential vulnerabilities. The objective is to aid translation to WSL for detecting the vulnerabilities and to enable local checking of inter-procedural properties.

(b) ASM2WSL

This module allows for the translation from assembly language to WSL. A sophisticated Assembler parser is used to capture the entire functionality of the Assembler code. This is then automatically converted to intermediate Wide Spectrum Language (WSL) designed specifically to support code transformation. In this module the extension of the existing translation of ASM to WSL, from assembly x86 16 bit to 32 bit focusing on the behaviour and model Assembly code to WSL action system code in certain semantic behaviour has been achieved. To extend the translation to 32 bit there are more registers and more operations that are supported. This will require the addition of more variables to the "virtual processor" to provide more considerations.
Appendix C: Additional Information on Process of Vulnerability Detection

related to partial access of the registers (i.e. al and ah being the lower and higher part of ax; there are more of these with EAX in 32bit).

The translator translates each instruction into a block of WSL code (in an action system). A complete translation of Assembler to WSL taking into consideration the possibility that the address passed into the module via a parameter is actually the address of the part of the code of the module. When data is written to this address, the module code is overwritten and changes its behaviour. **Under these conditions**, a complete translation can hardly do better than translate the code to a sequence of bytes which is "interpreted" as the program executes. Such translations are arguably more accurate than others. A description of the Assembly Translator Specifications is presented below:

**Adaptive Assembly Translator Specifications**

- Translates a subset of x86 assembly to WSL.
- Basically a line by line translator
- Focus is on translating all aspects.
- We work with a “virtual” processor
- All processor registers are local variables
- Low and High parts of registers implemented with additional operations
- Flags are variables too
- Overflow variable, needed for 8/16 bits
- Labels – Action system names
- Stack – a list
- Some special macros are recognized and translated directly
- procedures – nested Action systems

**Listing C-1 Adaptive Assembly Translator Specifications**

The buffer overflow involves overwriting part of the compiled code of the module. The translated WSL code can detect the overflow and then call a "placeholder" function. This is one of the most important analysis that is required in order to mark the vulnerabilities. However, if the "placeholder" function has not been called, then it can be proven or adjudged that buffer overflow does not occur. On the other hand, if it is established that the placeholder could be called under certain conditions, then it could be said that a security flaw in the code has been detected and needs to be addressed. See example below.

```plaintext
If "placeholder" function never be called then
   (Buffer overflow does not occur ;)
Else if {"placeholder" be called then
   We found Security flaw ;)
```

Shown below is the algorithm for the translator detailing step by step procedures.

**Input**: Source program
Appendix C: Additional Information on Process of Vulnerability Detection

---

**The Adaptive Process of Translator**

1. Apply Lexical Analyzer (Scanner) on source code (converting a sequence of characters into a sequence of tokens by removing any whitespace or comments in the source code)
2. Apply Syntax/Semantic Analyzer (Parser) on output from 1
3. Takes the input from a lexical analyzer in the form of token streams and checks whether the token stream meets the grammatical specification of the language and generates the syntax tree
4. Using taint analysis initialize all variables as Safe
5. Find functions or expression that may be an untrusted.
6. Check functions or expression. If a tainted value is used, mark the result of the functions or expression as Vulnerable.
7. Repeat step 5 until the end of the code.
8. Find all vulnerable functions and report it as vulnerability.
9. Intermediate WSL code Generator
10. Convert the token from c to WSL
11. Intermediate code Optimizer
12. Delete token that don’t effect WSL code
13. Final WSL Code Generator
14. Report vulnerable functions as vulnerability in WSL.

---

**Listing C- 2 Adaptive Process of Translator**

**Output:** WSL. Code with vulnerability labelling

**Part 2**

(a) **WSL**

Includes both low-level programming constructs and high-level abstract specifications within a single language. Such a language forms an ideal tool for developing methods for formal program development, and also for formal reverse engineering, because the proof that a program correctly implements a specification, or that a specification correctly captures the behavior of a program, can be achieved by means of semantic-preserving transformations in the language. Translating assembly programs to WSL provides options to: generate call diagrams for easier understanding of original code; automatically transform the code to much simpler versions; optionally to manually tweak the results with more transformations; facilitate easy vulnerability detection analysis. Against this backdrop, it is important to use the transformation engine to simplify the code to get effective and optimized code for better understanding.

(b) **Transformation Engine**

- **Transformation of the WSL:** Once the entire functionality of the Assembler code has been replicated within WSL, a series of sophisticated code transformations are automatically applied to the code to restructure and simplify the code to its optimum logical state. Afterwards, the vulnerability detection method after the migration and
Appendix C: Additional Information on Process of Vulnerability Detection

conversation can then be put into use. In order to apply transformations (i.e. optimizer) to convert the WSL action system code to WSL, there are many method of the transformation that could be adopted. However, in the current work specific transformation rules are selected that can provide the pattern of the code that is being targeted.

- **Optimization**: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations procedures include inline expansion, dead code elimination, constant propagation, loop transformation, register allocation and even automatic parallelization. This phase is responsible for transformation of the source program from one form to other.

- **Transformation Algorithm for Restructuring Action System**

Given that the source code are re-engineered, the focus is to absorb/merge the statements, remove redundant variables/flags, rewrite the loops, rearrange (i.e. by taking statements out of loops) and use constant propagation to reduce statements that are noteffective. By looking at examples of real programs, a number of common scenarios have been identified in which more complex transformation strategies can be employed beneficially. Since transformations are written as programs in MetaWSL, it is important to determine transformation rules for the Action system that has been used to reach the target as illustrated below:

**Expertise Rule (Action System)** The program transformation selected for a given action system should implement the heuristics for restructuring action systems. The restructuring steps can include the following steps.

1. Delete unreachable code;
2. Remove the tail recursion in an action which calls: by introducing a double-nested DO ... OD loop and replacing the self-calls by exits. Further transformation are then attempted to reduce the double loop to a single loop;
3. Simplify all IF Statements which contain calls;
4. Simplify action bodies to merge calls and remove recursion;
5. Eliminate actions which are only called once;
6. Shrink the action by creating procedures from blocks of code;
7. Remove the last action;

The set of operations has been integrated in a unified program transformation, known as Collapse_Action_System. Therefore, if the object of the program is an action system, i.e.,
Appendix C: Additional Information on Process of Vulnerability Detection

the state function is $@ST(@I) = T_A_S$, the candidate of applicable transformation can be Collapse_Action_System only. This expert rule is suitable for the program containing action system for the targets which need to collapse action system.

**Expertise Rule (Merging Similar Statements)** Similar statements can be merged and converted to a nested statement in the following scenarios.

1. Two non-nested similar IF statements can be merged and converted to a nested IF statement and taking the common code out of the two branches of the outer IF;
2. If the statements appear at the end of a loop and also just before the loop, then loop inversion can be applied to merge the two copies of the statement.

As for rule (i), for example, two copies "c := 1" can be combined by merging the two IF statements as follows.

<table>
<thead>
<tr>
<th>Original</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF a= 1 THEN</td>
<td>IF a = 1 V b = 2 THEN</td>
</tr>
<tr>
<td>c := 1; exit(2) FI;</td>
<td>THEN c := 1;</td>
</tr>
<tr>
<td>IF b = 2 THEN</td>
<td>IF a= 1 THEN exit(2) else exit(1) FI</td>
</tr>
<tr>
<td>c := 1; exit(1);</td>
<td>ELSE d := 3 FI;</td>
</tr>
<tr>
<td>ELSE d := 3 FI;</td>
<td></td>
</tr>
</tbody>
</table>

Below the example is for (ii)

<table>
<thead>
<tr>
<th>Original</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:= 1;</td>
<td>DO</td>
</tr>
<tr>
<td>DO</td>
<td>a:= 1;</td>
</tr>
<tr>
<td>b := 1;</td>
<td>DO b :=1;</td>
</tr>
<tr>
<td>IF c = 1 THEN a:= 1;</td>
<td>IF c = 1 THEN a :=1;</td>
</tr>
<tr>
<td>ELSE d := 1;</td>
<td>ELSE d := 1; IF; OD;</td>
</tr>
<tr>
<td>FI;</td>
<td>OD</td>
</tr>
<tr>
<td></td>
<td>ELSE d := 3 FI;</td>
</tr>
</tbody>
</table>

**Figure C-1 Program Transformation**

In the implementation, the 'absorb' transformations and the 'merge' transformations can be regarded as the candidates.

**Program Transformation Algorithm Steps**

The steps involved in the program transformation algorithm are illustrated below:
### Table C-1 Purpose of Transformation Rules

<table>
<thead>
<tr>
<th>Transformation Rules</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplify_Action_System</td>
<td>Simplify action system will attempt to remove actions and calls from an action system by successively applying simplified transformations. As many of the actions as possible will be eliminated without making the program significantly larger.</td>
</tr>
<tr>
<td>Collapse_Action_System</td>
<td>Collapse action system will use simplifications and substitution to transform an action system into a sequence of statements, possibly inside a DO loop.</td>
</tr>
<tr>
<td>Constant_Propagation</td>
<td>Constant Propagation finds assignments of constants to variables in the selected item and propagates the values through the selected item (replacing variables in expressions by the appropriate values)</td>
</tr>
<tr>
<td>Remove_Redundant_Vars</td>
<td>Remove Redundant Vars takes out as many local variables as possible from the selected VAR structure. If they can all be taken out, the VAR is replaced by its (possibly modified) body.</td>
</tr>
<tr>
<td>Delete_All_Redundant</td>
<td>Delete All Redundant searches for redundant statements and deletes all the ones it finds. A statement is 'Redundant' if it calls nothing external and the variables it modifies will all be assigned again before their values are accessed.</td>
</tr>
<tr>
<td>Delete_Item</td>
<td>This transformation will delete a program item that is redundant or unreachable</td>
</tr>
<tr>
<td>Simplify</td>
<td>This transformation will simplify any component as fully as possible.</td>
</tr>
<tr>
<td>Flag_Removal</td>
<td>Attempt to remove references to flag variables</td>
</tr>
<tr>
<td>Recursion_To_Loop</td>
<td>Remove Recursion in Action to replace the body of a recursive action if possible by an equivalent loop structure.</td>
</tr>
<tr>
<td>Floop_To_While</td>
<td>Convert a suitable DO...OD loop to a While loop</td>
</tr>
</tbody>
</table>

The above illustrations show the transformation rules that have been chosen for the conversion of action system in order to reach the desired target. More information on these set of rules are well documented in the WSL Reference Manual. In the above example, the source code for the FermaT “Simplify” transformation is very simple given that it just calls the @Simplify function on the current item, then it invokes the Simplify Item transformation on each component statement for which it is valid, then it invokes Delete Item transformation on every component statement for which it is invalid (other than assertions and comments). Finally it deletes SKIP statements within the current item:
Appendix C: Additional Information on Process of Vulnerability Detection

MW_PROC @Simplify_Code(Data) ==
@Paste_Over(@Simplify(@I, @Budget));
FOREACH Statement DO
  IF @Cs?(@I)
    THEN IF @Trans?(TR_Simplify_Item) THEN @Trans(TR_Simplify_Item, "") FI FI;
  IF @ST(@I) <> T_Comment AND @ST(@I) <> T_Assert AND @Trans?(TR_Delete_Item)
    THEN @Trans(TR_Delete_Item, "") FI OD;
  IF @Trans?(TR_Delete_All_Skips) THEN @Trans(TR_Delete_All_Skips, "") FI ;

Deleting a comment is always a valid transformation, but should not be carried out unless explicitly selected by the user. All the real work of FermaT Simplify is carried out by the @Simplify function. This takes a syntactic item and a ‘budget’ (an integer value which indicates how much effort should be expended in trying to simplify the item) and returns a new item. The requirements for this expression and condition simplifier were as follows. Below is a part of the code when it is built into the algorithm in META-WSL to be applied for the WSL code and transform the code based on the developer’s desire.

MWPROC @Process_Prog() ==
@Trans(TR_Constant_Propagation, "");
FOREACH Statement DO
  IF @ST(@I) = T_A_S THEN
    C: "don’t need to test for this, works for T_A_S";
    @Trans(TR_Simplify_Action_System, "");
    IF @Trans?(TR_Collapse_Action_System) THEN
      @Trans(TR_Collapse_Action_System, "");
    FI;
  ELSIF @Trans?(TR_Removal_of_Multiple_Statements) THEN
    @Trans(TR_Removal_of_Multiple_Statements, "");
  ELSIF @ST(@I) = T_Skip THEN
    @Delete
  END;
OD;

C: "remove all the comments ";
FOREACH Statement DO
  IF @ST(@I) = T_Comment THEN
    @Delete
  FI
OD;
C: "Convert DO loops into WHILE loops";
FOREACH Statement DO
  IF @Trans?(TR_Floop_To_While) THEN
    @Trans(TR_Floop_To_While, "");
  FI
OD;
C: "Go back to the start, and remove redundant";
@GOTO(< >);
@Trans(TR_Delete_All_Redundant, "");
SKIP

181
Appendix C: Additional Information on Process of Vulnerability Detection

Listing C-3 Meta-WSL Code for Transformation

After a successful conversion of the code that has been translated from assembly to WSL action, the system is now ready for the analyses of the vulnerability code with clear code to ensure that code is safe and this will be demonstrated in the next phase.

Part 3

(a) Database

Log presents the analysis information and help with application debugging in order to provide audit trails for vulnerability detection. It also has the pattern that has been translated from C to WSL in the translation process. However, in the vulnerability analysis it has the pattern of the vulnerability. System Information is where all details of the analysis are saved and it helps the decision maker to make the effective decision. Security policy presents all the rules that help in detecting the potential vulnerability. Example of potential vulnerability sources include:

- If data is read from an untrusted source.
- If untrusted data is insufficiently validated.
- If untrusted data is used in a potentially vulnerable function or a language construct.
- If the function copies the contents from one to another variables.
- If string pointed by X has bigger size, while the size of Y is less.

An untrusted source is defined as any source of data that can be influenced or modified by an attacker. The list of untrusted sources is application and system specific, but for most WSL programs it will include command line arguments, environmental variables, files and stdin input. Different sources give an attacker varying degrees of control over the data that reaches the program. There might be restrictions on its size, format and valid characters. A conservative worst-case approximation is to assume that all data coming from an untrusted source is completely arbitrary. A system that does not allow user input of any kind would be free of software vulnerabilities, because it would always perform as designed. Short of waiting for a random failure, an attacker would have no way to influence the execution of the program. But most computer programs are designed to process user data. Since user interaction is necessary in these programs, they must be regarded as potentially vulnerable. Not every use of untrusted data results in a vulnerability. In the current work, a limited number of potentially vulnerable functions and language constructs have been identified as listed in Table C-2.
Appendix C: Additional Information on Process of Vulnerability Detection

### Table C-2 Potentially Vulnerable Functions and Language Constructs

<table>
<thead>
<tr>
<th>Name</th>
<th>Functions of WSL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory allocation</td>
<td>ARRAY( )</td>
<td>Vulnerable if the value of the first parameter is a result of an arithmetic overflow during addition or multiplication involving untrusted data.</td>
</tr>
<tr>
<td>String manipulation</td>
<td>X := Y SUBSTR String copy</td>
<td>Vulnerable if the size of the destination buffer is smaller than the size of the string to be copied into it.</td>
</tr>
<tr>
<td>Memory copying</td>
<td>X[ ] := Y[ ] Out bounds</td>
<td>Vulnerable if the number of bytes to copy is greater than the size of the destination buffer.</td>
</tr>
<tr>
<td>Array access</td>
<td>Negative Index</td>
<td>Vulnerable if the array index is negative or greater than the size of the array.</td>
</tr>
<tr>
<td>Format string functions</td>
<td>PRINT( ) PRINFLUSH( )</td>
<td>Vulnerable if the format string contains un-trusted user data.</td>
</tr>
<tr>
<td>Insecure by design</td>
<td>@Read_Line(Standard _Input_Pot) String and read bugs</td>
<td>This function is always vulnerable.</td>
</tr>
</tbody>
</table>

Some of the above listed vulnerable functions are insecure by design, like the @Read_Line function, while others are vulnerable in a very specific set of circumstances. Most are safe if the untrusted data is properly validated. The degree of validation required depends on the type of data and its use. For example, when a string is used as an argument to string copy its length must be checked. If the same string is used in a PRINT() function, its length is irrelevant, but it should not contain any untrusted data. Vulnerability occurs when an execution path exists between a read from an untrusted source and a potentially vulnerable use with insufficient data validation. In this work, the approach to detecting these vulnerabilities is based on a combination of taint analysis and slicing via FermaT transformations in “WSL Vulnerability Analyser Technique” based on value size range in “WSL Memory Analyser Technique”. In the subsections that follow, a detailed description of these techniques is presented.

### Part 4

- **Program Slicing**

  The program slicing technique is used to trace through the code and extract all the dependant statements that can potentially affect the value of a given variable at a specific location of interest; known as slicing criterion <Statement location s, Variable name v>. Therefore, the sliced program is a subset of a full program which is used to isolate the fragment of the code for various reasons; the most common motives are for debugging, testing, maintaining and gaining
program comprehension Binkley (2007). The concept of slicing was initially introduced by Mark Weiser in 1979 as a debugging technique, which was later exploited in other areas by other researchers Tip (1994). Weiser (1979) defined a program slice $S$ as a reduced, executable program obtained from a program $P$ by removing statements, such that $S$ replicates part of the behavior of $P$ Ward (2003, 2007). There are many variants of program slicing available and can be separated in following ways: such as static/dynamic, amorphous, conditioned, syntactic/semantic and conditioned semantic slicing. Firstly, there are two main types of slicing techniques, a static slice, where just the source code is analysed and dynamic slice, where the tests are performed at run time with specific test cases. Secondly, these slices can be constructed syntactically (syntax preserving), semantically (syntactically equivalent) or both (amorphously). Finally, the program slicing can be performed in two directions, forward and backward. The forward slice inspects how the changes made to the value of a variable impacted the end result and the backward slice tracks what statements or variables have impacted the final outcome of the variable.

- **Forward and Backward Slicing**

  The forward slicing technique is used when there is the need to assess the impact of the change made at a given point in the program; a top down approach. A control-flow graph can be used to aid this searching computation. An example of a forward slicing is when the interest vulnerability variable value is changed to calculating the size for the vulnerability at the top of the function to see how it impacts the changes of the potential vulnerability size on the memory. In contrast, the backward slicing is a bottom up approach to determine how the end result of the program was achieved and what statements can or have influenced the final outcome; program dependence graph can be used for efficient computations. An example of backward slicing would be to trace through the source code upwards on how the size of the vulnerability value was calculated, so that one can see what path were used, what related functions were applied, which interest functions were applied and so on. Another example would be the stack trace feature in the integrated development environments (IDE’s) for Java programming language which specifies where exactly the system errors were thrown; enabling one to trace through the hierarchy of the calls to identify the potential cause of the problem.

- **Backwards Syntactic and Backwards Semantic Slicing**

  The term syntactic means that the program adheres to the programing language’s syntax which preserves the sequence of the program and can be compiled and executed, whereas the term semantic refers to the meaning of the program and if it performs in the way it was intended
Appendix C: Additional Information on Process of Vulnerability Detection

to, without worrying about preserving the syntax or a sequence of the statement. More formally, a syntactic slice of a program \( S \) is any program \( S' \) formed by modifying and removing other non-affecting statements from \( S \), then the original program \( S \) and sliced program \( S' \) are semantically equivalent. Therefore when performing backward syntactic slicing, we trace up the program life cycle and only delete the components (i.e. statements and predicates) that do not affect the end result of the variable we are interested in, whilst preserving the sequence and operational semantics of the program. Whereas, the backward semantic slicing is concerned with preserving the final state of the variable; which may result in changing the sequence of the program, refining the statements and restructuring the program but the final program is semantically equivalent to the original program. The WSL Vulnerability Analyser (WVA) was integrated with the WSL Slicing Transformation (WST) which helps to isolate the vulnerable code from the original source code after labelling the unsafe code. This will help in the next stage to check the side effect on the memory with less time consumed for analysing with boundary by integrating with WSL Memory Analyser (WMA).

Definition of the slicing relation as a WSL condition, to define the slicing relation as a WSL condition, we need a way to express conditions such as

\[
\forall (V \setminus Y).WP(S,R),
\]

Where \( V \) is the initial state space and \( Y \) is another set of variables. Define the formula

\[
wpa (y, I, v),
\]

which is true when

\[
\forall (\tilde{v} \setminus \tilde{y}).WP(\tilde{I}, \text{true})
\]

Is true:

\[
\begin{align*}
    wpa (y, I, v) & \equiv \text{DF} \bigvee_{0 \leq I < \omega} (y = y_k) \land \bigvee_{0 \leq n < \omega} (v = v_n) \\
    & \land \bigwedge_{0 \leq k, n < \omega} ((I = \tilde{S}_n) \land v = v_n \land y = y_k) \\
    & \iff \forall (V \setminus V_\tilde{y}).WP(S_\text{true}).
\end{align*}
\]

For our running example, recall that \( V_3 \) is the set \( \{c\} \) and \( v_3 \) is the representation of \( V_3 \). Then,

\[
wpa (v_3, \tilde{S}_2, v) \iff \forall_a b.WP(S_3, \text{true}) \iff \text{true}.
\]
Appendix C: Additional Information on Process of Vulnerability Detection

Define the formula $\text{wpa}_{\phi}(y, I, u, w)$, which is true when

$$\forall (v \setminus y).\text{WP}(I, w, w')$$

is true: $\text{wpa}_{\phi}(y, I, u, w) = \text{DF}$

Fins $(I, u, w) \wedge \bigvee_{0 < k < \omega} (y = u_k)$

$\wedge \bigwedge_{0 < b, m, n < \omega} ((I = S_m \wedge u = v_n \wedge u = v_n \wedge y = u_k) \Rightarrow \forall (V_n \setminus V_k).\text{WP} (S_m, \overline{V_m} \neq \overline{V_{m,n}}))$.

For our running example:

$\text{wpa}_{\phi}(v_3, S_3, v_3, v_3)$

$\iff \forall a, b.\text{WP}(S_2, c \neq d')$

$\iff \forall a, b.\text{WP}(\text{skip}, \text{skip}; c := 3, c \neq d')$

$\iff \forall a, b.(3 \neq d')$

$\iff 3 \neq d'$.

For more detail about the formula see Ward and Hussein Zedan (2011)
Appendix D: Prototype of Vulnerability Detection

Appendix D: Prototype of Vulnerability Detection

Development Tool Prototype Design of FermaT Wide Spectrum 2015 Express

Figure D-1 Development Tool Prototype Design of Vulnerability Detection

Classification of Software Vulnerabilities with requirement needed

Table D-1 Vulnerability Patterns

<table>
<thead>
<tr>
<th>Functions of C</th>
<th>Functions of WSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *_malloc(size_t size)</td>
<td>ARRAY(1, Size)</td>
</tr>
<tr>
<td>void *_calloc(size_t nitems, size_t size)</td>
<td>ARRAY(No. of element, Size of element)</td>
</tr>
<tr>
<td>void *_realloc(void *ptr, size_t size)</td>
<td>ARRAY(1, Size)</td>
</tr>
<tr>
<td>char *strncpy(char *dest, const char *src)</td>
<td>X := Y</td>
</tr>
<tr>
<td>char *strcat(char *dest, const char *src)</td>
<td>s1 ++ s2</td>
</tr>
<tr>
<td>void *memcpy(void *str1, const void *str2, size_t n)</td>
<td>SUBSTR(str1,n)</td>
</tr>
<tr>
<td>void *memmove(void *str1, const void *str2, size_t n)</td>
<td>SUBSTR(str, l)</td>
</tr>
<tr>
<td>type arrayName [arraySize];</td>
<td>ARRAY(No. of element, Size of element)</td>
</tr>
<tr>
<td>int printf(const char *format, ...)</td>
<td>PRINT(@type(s))</td>
</tr>
<tr>
<td>int sprintf(char *str, const char *format, ...)</td>
<td>@Read_Line(Standard_Input_Port)</td>
</tr>
<tr>
<td>char *gets(char *str)</td>
<td></td>
</tr>
<tr>
<td>int scanf(const char *format, ...)</td>
<td></td>
</tr>
</tbody>
</table>

The System requirements and choice of programming language.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic string buffer overflow</td>
<td>The string s is read from an untrusted source and its length is controlled by the attacker. The string is copied into a buffer of fixed size using the string copy function. If the length of the string exceeds the size of the destination buffer, the string copy function will overwrite data past the end of the buffer.</td>
</tr>
<tr>
<td>Memory copy buffer overflow</td>
<td>The integer n is read from an untrusted source and its value is controlled by the attacker. No input validation is performed before n is used as the third argument of memory copy function. If the attacker chooses a value of n bigger than the size of the destination buffer, the memory copy function will overwrite data past the end of the buffer.</td>
</tr>
<tr>
<td>Format string bug</td>
<td>The string s is read from an untrusted source and its contents are controlled by the attacker. The string is used as a format string in one of the functions of the print commands family. By including special format specifiers like “%n” in the format string, the attacker can trick print command into writing to an arbitrary memory location.</td>
</tr>
<tr>
<td>Out of bounds array access</td>
<td>The integer n is read from an untrusted source and its value is controlled by the attacker. No input validation is performed before it is used as an array index. By choosing a specific value of n, the attacker can access any memory location on the system. Depending on the program, this can give the attacker the ability to read or write to an arbitrary memory location, which is easily exploitable.</td>
</tr>
<tr>
<td>Out of bounds array access with a negative array index</td>
<td>The signed integer n is read from an untrusted source and its value is controlled by the attacker. The program validates the integer n by making sure it is less than the size of the array and then uses it as an array index. If the attacker supplies a negative value, the condition of the if statement will be satisfied. A negative array index gives the attacker the ability to write to an arbitrary memory location, as long as it has a lower memory address than the array.</td>
</tr>
<tr>
<td>Arithmetic overflow when allocating an array of objects</td>
<td>The integer n is read from an untrusted source and its value is controlled by the attacker. multiply n by a constant. By choosing a specific large value of n, the attacker can cause an arithmetic overflow and allocate 0 bytes for the array. Any subsequent use of the array will access data which has not been allocated.</td>
</tr>
<tr>
<td>Signed to unsigned conversion.</td>
<td>The signed integer n is read from an untrusted source and its value is controlled by the attacker. If the attacker supplies a negative value, the condition of the if statement will be satisfied and the memory copy function will be called. The third parameter to memory copy is declared as an unsigned integer and the compiler will implicitly cast n to an unsigned integer. By choosing a specific negative value of n, the result of the cast will be a positive number greater than 10. The memory copy function will overwrite data past the end of the destination buffer.</td>
</tr>
<tr>
<td>Scanf buffer overflow</td>
<td>A function of the scan family is used to read user input into a fixed size buffer. If the “%s” format specifier is used, scan will read user input until either whitespace or the end of file is reached. By supplying more data than the buffer can hold, an attacker can overwrite data past the end of the array.</td>
</tr>
<tr>
<td>Gets buffer overflow</td>
<td>The gets function is used to read user input into a fixed size buffer. By supplying more data than the buffer can hold, an attacker can overwrite data past the end of the buffer.</td>
</tr>
</tbody>
</table>
Appendix D: Prototype of Vulnerability Detection

Table D-3 the Common Weakness Enumeration (CWE)

<table>
<thead>
<tr>
<th>No.</th>
<th>Website</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="https://www.owasp.org/index.php/Buffer_Overflows">https://www.owasp.org/index.php/Buffer_Overflows</a></td>
<td>(CWE-1) – (CWE-2)</td>
</tr>
<tr>
<td>2</td>
<td><a href="http://www.thegekstuff.com/2013/06/buffer-overflow/">http://www.thegekstuff.com/2013/06/buffer-overflow/</a></td>
<td>(CWE-3)</td>
</tr>
<tr>
<td>4</td>
<td><a href="http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1.html">http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1.html</a></td>
<td>(CWE-9)</td>
</tr>
<tr>
<td>5</td>
<td><a href="https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/">https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/</a></td>
<td>(CWE-10)</td>
</tr>
<tr>
<td>7</td>
<td><a href="http://stackoverflow.com/questions/7344226/buffer-overflow-attack">http://stackoverflow.com/questions/7344226/buffer-overflow-attack</a></td>
<td>(CWE-12) – (CWE-13)</td>
</tr>
<tr>
<td>8</td>
<td><a href="http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Buffer_Overflow/">http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Buffer_Overflow/</a></td>
<td>(CWE-14) – (CWE-16)</td>
</tr>
<tr>
<td>9</td>
<td><a href="http://insecure.org/stf/smashstack.html">http://insecure.org/stf/smashstack.html</a></td>
<td>(CWE-17) – (CWE-21)</td>
</tr>
<tr>
<td>11</td>
<td><a href="http://c2.com/cgi/wiki:CeeLanguageAndBufferOverflows">http://c2.com/cgi/wiki:CeeLanguageAndBufferOverflows</a></td>
<td>(CWE-23)</td>
</tr>
</tbody>
</table>

Proposed Solution

There exists a pattern that is common in some classes of vulnerabilities that has been identified in this work. These classes were translated into assembly language using the assembler of Intel in Window 7 in order to trace and detect the source of the vulnerability. The traced vulnerabilities emanated from the path of execution based on certain critical features including the reading of data from a source that is adjudged untrusted; insufficient validation of untrusted data and the use of untrusted data within functions or language constructs that are potentially vulnerable; lack of boundary checking and lack of proper tracking and tracing the relative code of vulnerability within the program. In order to address the aforementioned problems, a proposed solution based on the framework schematically illustrated in Figure 5-2 is adopted. A description of the abbreviations used in Figure 5-2 is provided in Table 5-3.
Appendix D: Prototype of Vulnerability Detection

Figure D-2 Simple Illustration of Vulnerability Detection Framework Adopted in this Work

Table D-4 The Implementation Process Abbreviation.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2WSL</td>
<td>Translation from C Language to Wide Spectrum Language</td>
</tr>
<tr>
<td>C2ASM</td>
<td>Translation from C Language to Assembly Language</td>
</tr>
<tr>
<td>ASM2WSL</td>
<td>Translation from Assembly Language to Wide Spectrum Language</td>
</tr>
<tr>
<td>WSL</td>
<td>Wide Spectrum Language</td>
</tr>
<tr>
<td>WVAT</td>
<td>WSL Vulnerability Analysis Technique</td>
</tr>
<tr>
<td>WMAT</td>
<td>WSL Memory Analysis Technique</td>
</tr>
<tr>
<td>FME</td>
<td>FermaT Maintenance Environment</td>
</tr>
<tr>
<td>FST</td>
<td>FermaT Slicing Transformation</td>
</tr>
<tr>
<td>TA</td>
<td>Taint Analysis</td>
</tr>
<tr>
<td>VR</td>
<td>Value Range</td>
</tr>
<tr>
<td>CG</td>
<td>Call Graphs</td>
</tr>
<tr>
<td>M</td>
<td>Matrices</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>FSSA</td>
<td>FermaT Single Static Analysis</td>
</tr>
</tbody>
</table>

Figure D-3 Simple Illustration of Vulnerability Detection Framework Adopted in this Work
Appendix D: Prototype of Vulnerability Detection

Figure D-4 Display of Parser Log File.

C2WSL Log Message which shows the success of translation is depicted in Figure D-6 below.

Figure D-5 Illustration of a Successful Translation.

The output of the translation process is depicted as indicated below.

Table D-5 The Output of the Translation Process.

```plaintext
VAR <buff := ARRAY(15,0), pass:=0>,
PRINFLUSH (" Enter the password: ");
COMMENT:"safe from c code analysis**",
buff := @Read_Line(Standard_Input_Port);
COMMENT:"vunl by gets from c code analysis**",
Input := "thegeekstuff";
COMMENT:"vunl by strcm from c code analysis **",
IF SLENGTH(buff) <= SLENGTH(Input) AND buff <> Input THEN
PRINFLUSH ("Wrong Password ");
COMMENT:"safe from c code analysis **",
ELSE
PRINFLUSH ("Correct Password ");
COMMENT:"safe from c code analysis****",
pass:=1;
FI;
IF pass=1 THEN
PRINFLUSH ("*** Root privileges given to the user ***");
COMMENT:"vulnerable**",
FI;
SKIP
ENDVAR
```

The above is WSL code and been translated from C source program the been discussed earlier by C2WSL tool.
Appendix D: Prototype of Vulnerability Detection

The only way to ensure that the code has been translated accordingly is for it to behave in the same manner as the original code of the C program. This can be ascertained by checking the output of the execution of the WSL code in comparison with the input. The validation code has been implemented in the FME program as shown in Figure D-6.

![Figure D-6 Running the code an input “this string is too long”](image1)

Figure D-6 Running the code an input “this string is too long”.

Figure D-8 shows that after running the code an input “this string is too long” is entered and the output shown has similar results suggesting that the program has the same behaviour with the same output.

![Figure D-7 The Output of Hacked Code after the Translation](image2)

Figure D-7 The Output of Hacked Code after the Translation.

192
Appendix D: Prototype of Vulnerability Detection

Figure D-8: show the output of hacked code after the translation. The program has been tested with a variety of inputs, yielding correct results in every instance. More information and details about the program is illustrated in section 5.4.1

![Call Graph of WSL Code.](image)

The call graph show 1 output from the source code which indicates the potential vulnerability might exist. The following Figure 5-11 is the dataflow of the assembly code and the call graph. The “exe file” of the C program was adopted for the decompiling based on the “Retargetable Decompiler tool” which is available via “https://retdec.com/decompilation” to generate the assembly code and associated diagrams. Figure 5.10 shows the software result with the verification of the generation output “[OK]” from which more details of the binary executable files can be determined.

The following graphs (Figure D-9) shows that the main functions of the C program detected all from the “exe file” by the decompile and the vulnerabilities functions of C code can be seen clearly. However, the tool does not support the vulnerability detection functions of the low level language due to lack of traceability functions. As such, it cannot reveal much information about the vulnerable code and even if it does it will be more difficult to understand the complexity of the assemble code and it will be time consuming as well. Additionally, there is no transformation engine to optimize the code in order to help the developers in future to refine the code easily and make transformation of insecure codes to secure codes thereby mitigating potential vulnerability. The “Call Graph” shows the call functions in the assembly code and the “Control-flow Graph” shows all paths that traverse through a program during its execution. These graphs are generated using C Program based on the “binary executable file” within the retdec tool and
Appendix D: Prototype of Vulnerability Detection

provides excellent results. This suggests that the original codes can still be retrieved even if it is only the binary executable file. This retrieval capability provides a great deal of usefulness in the methods proposed in the current work.

![Call Graph and Control-Flow Graph](image-url)

The ASM2WSL tool translates the assembly code as block code for every call functions and it split the code as blocks to make it easy to understand and establish a relationship from every block to each other through the “Call Graphs”. In addition, it will take all the registers and flags
Appendix D: Prototype of Vulnerability Detection

declaration and place them on the top of the code. Figure D-9 shows the call graphs from the “STRL Visualisation Engine”.

Figure D-10 shows start point as “(Start)” and end point as “Z” and the rest is call functions inside the Acton system and they are 10. It shows the calling link of the relationship between each other and in this level the component of the code can be traced to help in the restructuring of the code by using the transformation engine which will be simplify the code. From Action System Call Graphs, the vulnerability functions can be traced using functions such as “_strcmp and _gets” and from Function/Procedure Call Graphs we can see the function “@Read_Line” which both of the diagram indicates the potential vulnerability. The transformation process is demonstrated as indicated below. The program was after the translation from assembly to WSL Action System and the result, yielding similar results as expected.

![Figure D-10 The Output of WSL Code after the Translation](image)
Appendix D: Prototype of Vulnerability Detection

Figure D-11 The Process of Transformation as in (a), (b), (c) and (d).
Appendix D: Prototype of Vulnerability Detection

The above screen shows the analysis of every step from the transformations. First calculating the number of the action in the system, which is 10 actions then, it does the following steps for the code sample:

**Table D- 6 Process of Transformation**

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplifying the result 10 actions</td>
<td>Simplifying action bodies</td>
</tr>
<tr>
<td>Deleting unreachable code</td>
<td>Eliminating actions which are only called once</td>
</tr>
<tr>
<td>Leave_Alone_Names = 10 actions</td>
<td></td>
</tr>
<tr>
<td>Simplifying conditional statements</td>
<td></td>
</tr>
<tr>
<td>Eliminating actions which are only called once</td>
<td></td>
</tr>
<tr>
<td>Finding elementary actions</td>
<td></td>
</tr>
<tr>
<td>Removing elementary actions</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th>Step 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>REITERATING...</td>
<td>Deleting unreachable code</td>
</tr>
<tr>
<td>Simplifying conditional statements:</td>
<td>CAS: Finding actions which only call Z...</td>
</tr>
<tr>
<td>Finding elementary actions</td>
<td></td>
</tr>
<tr>
<td>Removing elementary actions</td>
<td>Creating procedure 'l4', size = 1, tv = 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 5</th>
<th>Step 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching for other copies of body:</td>
<td>Simplifying the result</td>
</tr>
<tr>
<td>Using absorption to reduce loop body...a</td>
<td>Constant Propagation effort = 0 budget = 25600</td>
</tr>
<tr>
<td>Top_Level_Remove:</td>
<td>Call Budget = 25600</td>
</tr>
<tr>
<td>(c_e_m_p overflow flag_c flag_a flag_p flag_s flag_t flag_i fl ag_d flag_o ces ecc edss esi edx ecx es cs ds ss sp bp di sx cx bx ax)</td>
<td>DSECTs =</td>
</tr>
<tr>
<td>Constant Propagation effort = 0 budget = 25600</td>
<td>xBODY NOT FOUND for proc: l4</td>
</tr>
<tr>
<td>Call Budget = 25600</td>
<td>xBODY NOT FOUND for proc: l4</td>
</tr>
<tr>
<td>DSECTs =</td>
<td>BODY NOT FOUND for proc: l4</td>
</tr>
<tr>
<td>xBODY NOT FOUND for proc: l4</td>
<td>BODY NOT FOUND for proc: l4</td>
</tr>
<tr>
<td>BODY NOT FOUND for proc: l4</td>
<td>BODY NOT FOUND for proc: l4</td>
</tr>
<tr>
<td>BODY NOT FOUND for proc: l4</td>
<td>Simplifying the result</td>
</tr>
<tr>
<td>Simplifying the result</td>
<td>Redundant variables are: (ebp esp flag_z)</td>
</tr>
<tr>
<td>Redundant variables are: ()</td>
<td>Redundant variables are: ()</td>
</tr>
<tr>
<td>Redundant variables are: (eax)</td>
<td>Redundant variables are: ()</td>
</tr>
</tbody>
</table>

Figure D- 12 The Call Graph After the Transformation.
Appendix D: Prototype of Vulnerability Detection

The call graph in Figure D-12 show 4 output from the source code 3 goes to l4 and 1 to the @Read_Line which indicate the potential vulnerability might be exist. Table D-7 shows the differences before and after the transformation and also provides a lists of the metrics for the raw WSL translation after automatic restructuring and simplifications of the transformations rules have been applied. The comparison between C2WSL and ASM2WSL.

Table D-7 Description of Key Metrics for Translations within WSL.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Raw WSL</th>
<th>Structured WSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statements</td>
<td>69</td>
<td>31</td>
</tr>
<tr>
<td>Expressions</td>
<td>124</td>
<td>66</td>
</tr>
<tr>
<td>McCabe</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Essential</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CFDF</td>
<td>94</td>
<td>33</td>
</tr>
<tr>
<td>Branch-Loop</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>Structural</td>
<td>1101</td>
<td>497</td>
</tr>
<tr>
<td>Time-execution</td>
<td>2163s</td>
<td>1416s</td>
</tr>
</tbody>
</table>

In this section, a comparison of the output from the both translator is presented to ensure that the translator works correctly in terms of the output of execution after the translation from C2WSL and ASM2WSL has already taken place whilst taking into account the metrics result and the call Graphs.

Table D-8 Description of Key Metrics for Translations within C2WSL and ASM2WSL

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Raw WSL</th>
<th>Structured WSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statements</td>
<td>69</td>
<td>31</td>
</tr>
<tr>
<td>Expressions</td>
<td>124</td>
<td>66</td>
</tr>
<tr>
<td>McCabe</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Essential</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CFDF</td>
<td>94</td>
<td>33</td>
</tr>
<tr>
<td>Branch-Loop</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>Structural</td>
<td>1101</td>
<td>497</td>
</tr>
<tr>
<td>Time-execution</td>
<td>2163s</td>
<td>1416s</td>
</tr>
</tbody>
</table>

As shown in Table D-5 above, the metrics show different number from each translator and the C2WSL has less numbers because it has been translated directly from C code which is already has a shorter lines of code. However, ASM2WSL has larger number of codes because the assembly code is already long enough. The output of both translator shows similar behaviour but the call graph reveals a slightly different diagram given that the assembly code entails more details. This difference constitutes an advantage for the next level of vulnerability detection.
Appendix D: Prototype of Vulnerability Detection

Figure D-13 Abstract Syntax Tree in WSL

To get a forwards slice, start the data string with ",f," followed by the variable. By default this will slice from the start of the selected section of code. For example, with the program:

```plaintext
sum := sum0;
prod := 1;
i := 1;
WHILE i <= n DO
    sum := sum + A[i];
    prod := prod * A[i];
i := i + 1 OD;
PRINT("sum = ", sum);
PRINT("prod = ", prod)
```

A syntactic slice on ",f,sum0" will produce:

```plaintext
sum := sum0;
WHILE i <= n DO sum := sum + A[i] OD;
PRINT("sum = ", sum);
PRINT("prod = ", prod)
```

Listing D-1 Example of Slicing
Appendix D: Prototype of Vulnerability Detection

Figure D- 14 Call Graph Showing how Syntactic Slicing was Applied within the Codes.

Semantic Slicing (stack_44)

```
stack_44 := @Read_Line(Standard_Input_Port)
```

show the semantic slice we applied
Appendix D: Prototype of Vulnerability Detection

Syntax Slicing Call Graph

Semantic Slicing Call Graph

Semantic Slicing (stack_15)

1. IF @Read_Line(Standard_Input_Port) = "thegeekstuff" THEN
2. stack_15 := "***Root privileges given to the user***"
3. ELSIF SLENGTH(@Read_Line(Standard_Input_Port)) <= 12 THEN
4. stack_15 := "Wrong Password"
5. ELSE
6. stack_15 := "***Root privileges given to the user***"
7. FI

Figure D-15 Call Graph after Slicing.

Table D-9 Vulnerability Slicing to Detect the Vulnerabilities.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Vuln_Slicing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statements</td>
<td>6</td>
</tr>
<tr>
<td>Expressions</td>
<td>13</td>
</tr>
<tr>
<td>McCabe</td>
<td>3</td>
</tr>
<tr>
<td>Essential</td>
<td>1</td>
</tr>
<tr>
<td>CFDF</td>
<td>9</td>
</tr>
<tr>
<td>Branch-Loop</td>
<td>0</td>
</tr>
<tr>
<td>Structural</td>
<td>100</td>
</tr>
<tr>
<td>Time- execution</td>
<td>783s</td>
</tr>
</tbody>
</table>

This is stored internally as a parse tree, as illustrated in Figure D-16.
Appendix D: Prototype of Vulnerability Detection

Figure D-16 Abstract Syntax Tree for the Vulnerable Code.

Figure D-17 Call Graph after the SSA Transformation Process.

Suspicious Variable Value Range (SVVR) formula is:

\[ SVVR = \frac{\text{Taint Analysis} + \text{Slicing}}{\text{Bound Checking}} \quad (0-1) \]

Dump heap = Variable.Size (Actual) - Variable.Size (Copied)
Appendix D: Prototype of Vulnerability Detection

Vulnerability Examples

The following are the examples which demonstrate the investigation of the potential vulnerability code.

Example (1)

The example below shows with the highlighted the differences between the two examples

<table>
<thead>
<tr>
<th>Vulnerability code in C</th>
<th>Non-Vulnerability code in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;stdio.h&gt;</td>
<td>#include &lt;stdio.h&gt;</td>
</tr>
<tr>
<td>#include &lt;string.h&gt;</td>
<td>#include &lt;string.h&gt;</td>
</tr>
<tr>
<td>int main(void)</td>
<td>int main(void)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>char str1[10];</td>
<td>char str1[10];</td>
</tr>
<tr>
<td>strcpy(str1,str2);</td>
<td>strcpy(str1,str2);</td>
</tr>
<tr>
<td>printf(str1);</td>
<td>printf(str1);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Below shows the assembly code that is the output of the disassembler from the binary file.

<table>
<thead>
<tr>
<th>Vulnerability code in assembly</th>
<th>Non-Vulnerability code in assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>_main:</td>
<td>_main:</td>
</tr>
<tr>
<td>LFB10:</td>
<td>LFB10:</td>
</tr>
<tr>
<td>push ebp</td>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>and esp, -16</td>
<td>and esp, -16</td>
</tr>
<tr>
<td>sub esp, 48</td>
<td>sub esp, 48</td>
</tr>
<tr>
<td>call _main</td>
<td>call _main</td>
</tr>
<tr>
<td>mov DWORD PTR [esp+23], 1684234849</td>
<td>mov DWORD PTR [esp+23], 1684234849</td>
</tr>
<tr>
<td>mov DWORD PTR [esp+27], 1751606885</td>
<td>mov DWORD PTR [esp+27], 1751606885</td>
</tr>
<tr>
<td>mov DWORD PTR [esp+31], 1818978921</td>
<td>mov DWORD PTR [esp+31], 1818978921</td>
</tr>
<tr>
<td>mov WORD PTR [esp+35], 28269</td>
<td>mov WORD PTR [esp+35], 28269</td>
</tr>
<tr>
<td>mov BYTE PTR [esp+37], 1</td>
<td>mov BYTE PTR [esp+37], 1</td>
</tr>
<tr>
<td>lea eax, [esp+23]</td>
<td>lea eax, [esp+23]</td>
</tr>
<tr>
<td>mov DWORD PTR [esp+4], eax</td>
<td>mov DWORD PTR [esp+4], eax</td>
</tr>
<tr>
<td>lea eax, [esp+38]</td>
<td>lea eax, [esp+38]</td>
</tr>
<tr>
<td>mov DWORD PTR [esp], eax</td>
<td>mov DWORD PTR [esp], eax</td>
</tr>
<tr>
<td>call _strncpy</td>
<td>call _strncpy</td>
</tr>
<tr>
<td>lea eax, [esp+38]</td>
<td>lea eax, [esp+38]</td>
</tr>
<tr>
<td>mov DWORD PTR [esp], eax</td>
<td>mov DWORD PTR [esp], eax</td>
</tr>
<tr>
<td>call _printf</td>
<td>call _printf</td>
</tr>
<tr>
<td>leave</td>
<td>leave</td>
</tr>
<tr>
<td>ret LFE10:</td>
<td>ret LFE10:</td>
</tr>
</tbody>
</table>

1. **sub esp, 48** size the has been reserved inside the memory
Appendix D: Prototype of Vulnerability Detection

2-  

\[ \text{mov} \quad \text{BYTE PTR [esp+37]}, 0 \]  
where we end saving the 14 byte then return 0 to address \[ \text{mov} \quad \text{DWORD PTR [esp+23]}, 1684234849 \]  

The following code show that we have 14 byte been reserved and stored the value

\[ \text{mov} \quad \text{DWORD PTR [esp+23]}, 1684234849 \]
\[ \text{mov} \quad \text{DWORD PTR [esp+27]}, 1751606885 \]
\[ \text{mov} \quad \text{DWORD PTR [esp+31]}, 1818978921 \]
\[ \text{mov} \quad \text{WORD PTR [esp+35]}, 28269 \]

3-  

\[ \text{lea} \quad \text{eax, [esp+38]} \]  
Load effective address 48 - 38 = 10 byte that is free local space and will be used

4-  

\[ \text{mov} \quad \text{DWORD PTR [esp+8]}, 10 \]  
the beginning address values is the values that been used in the code which may indicate to the size or something else. This value gives clue to analysis the code in term of the other addresses

5-  

\[ \text{call} \quad _\text{strncpy} \]  
is a string copy with specifying the size so seeing like 10 and 10 indicate that the size been specify is 10 and the size of reserved place inside the memory is also 10.

When we translate to WSL we take these to our consideration to identify the vulnerability. Every vulnerability might have different features so we actually gathered the different features from different example to train our analysis system to detect a variety of vulnerability.

Below you will find different features

Example (2)

<table>
<thead>
<tr>
<th>Vulnerability code in C</th>
<th>Non-Vulnerability code in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;stdio.h&gt;</td>
<td>#include &lt;stdio.h&gt;</td>
</tr>
<tr>
<td>#include &lt;string.h&gt;</td>
<td>#include &lt;string.h&gt;</td>
</tr>
<tr>
<td>int main(int argc, char **argv)</td>
<td>int main(int argc, char **argv)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>char buf[8];</td>
<td>char buf[8];</td>
</tr>
<tr>
<td>gets(buf);</td>
<td>f gets(buf, sizeof(buf), stdin);</td>
</tr>
<tr>
<td>printf(&quot;%s\n&quot;, buf);</td>
<td>printf(&quot;%s\n&quot;, buf);</td>
</tr>
<tr>
<td>return 0;</td>
<td>return 0;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
### Appendix D: Prototype of Vulnerability Detection

<table>
<thead>
<tr>
<th>Vulnerability code in assembly</th>
<th>Non-Vulnerability code in assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_main:</code></td>
<td><code>_main:</code></td>
</tr>
<tr>
<td>push ebp</td>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>and esp, -16</td>
<td>and esp, -16</td>
</tr>
<tr>
<td>sub esp, 32</td>
<td>sub esp, 32</td>
</tr>
<tr>
<td>call ___main</td>
<td>call ___main</td>
</tr>
<tr>
<td>lea eax, [esp+24]</td>
<td>lea eax, [esp+24]</td>
</tr>
<tr>
<td>mov DWORD PTR [esp], eax</td>
<td>mov DWORD PTR [esp], eax</td>
</tr>
<tr>
<td>call ___gets</td>
<td>call ___gets</td>
</tr>
<tr>
<td>mov eax, 0</td>
<td>mov eax, 0</td>
</tr>
<tr>
<td>leave</td>
<td>leave</td>
</tr>
<tr>
<td>ret</td>
<td>ret</td>
</tr>
</tbody>
</table>

| LFE10:                          | LFE10:                            |

The `mov eax, DWORD PTR ___imp___iob` means we have outsourced input which will be saved in the following address. `mov DWORD PTR [esp+8], eax`

### Example (3)

<table>
<thead>
<tr>
<th>Vulnerability code in C</th>
<th>Non-Vulnerability code in C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;stdio.h&gt;</code></td>
<td><code>#include &lt;stdio.h&gt;</code></td>
</tr>
<tr>
<td><code>#include &lt;stdlib.h&gt;</code></td>
<td><code>#include &lt;stdlib.h&gt;</code></td>
</tr>
<tr>
<td><code>enum {BUFFER_SIZE = 10};</code></td>
<td><code>enum {BUFFER_SIZE = 10};</code></td>
</tr>
<tr>
<td><code>int main() {</code></td>
<td><code>int main() {</code></td>
</tr>
<tr>
<td><code>char buffer[BUFFER_SIZE];</code></td>
<td><code>char buffer[BUFFER_SIZE];</code></td>
</tr>
<tr>
<td><code>int check = 0;</code></td>
<td><code>int check = 0;</code></td>
</tr>
<tr>
<td><code>sprintf(buffer, &quot;%s&quot;, &quot;This string is too long!&quot;);</code></td>
<td><code>printf(buffer, &quot;%s&quot;, &quot;This string is too long!&quot;);</code></td>
</tr>
<tr>
<td><code>printf(&quot;check: %d&quot;, check);</code></td>
<td><code>printf(&quot;check: %d&quot;, check);</code></td>
</tr>
<tr>
<td><code>return EXIT_SUCCESS;</code></td>
<td><code>return EXIT_SUCCESS;</code></td>
</tr>
</tbody>
</table>

```c
#include <stdio.h>
#include <stdlib.h>
enum {BUFFER_SIZE = 10};
int main() {
    char buffer[BUFFER_SIZE];
    int check = 0;
    sprintf(buffer, "%s", "This string is too long!");
    printf("check: %d", check);
    return EXIT_SUCCESS;
}
```
Appendix D: Prototype of Vulnerability Detection

<table>
<thead>
<tr>
<th>Vulnerability code in assembly</th>
<th>Non-Vulnerability code in assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.ascii &quot;check: %d\0&quot;</code></td>
<td><code>.ascii &quot;This string is too long!\0&quot;</code></td>
</tr>
<tr>
<td><code>_main:</code></td>
<td><code>_main:</code></td>
</tr>
<tr>
<td>LFB12:</td>
<td>LFB12:</td>
</tr>
<tr>
<td><code>push ebp</code></td>
<td><code>push ebp</code></td>
</tr>
<tr>
<td><code>mov ebp, esp</code></td>
<td><code>mov ebp, esp</code></td>
</tr>
<tr>
<td><code>and esp, -16</code></td>
<td><code>and esp, -16</code></td>
</tr>
<tr>
<td><code>sub esp, 32</code></td>
<td><code>sub esp, 32</code></td>
</tr>
<tr>
<td><code>call ___main</code></td>
<td><code>call ___main</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [esp+28], 0</code></td>
<td><code>mov DWORD PTR [esp+28], 0</code></td>
</tr>
<tr>
<td><code>lea eax, [esp+18]</code></td>
<td><code>lea eax, [esp+18]</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [eax], 1936287828</code></td>
<td><code>mov DWORD PTR [eax], 1936287828</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [eax+4], 1920234272</code></td>
<td><code>mov DWORD PTR [eax+4], 1920234272</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [eax+8], 543649385</code></td>
<td><code>mov DWORD PTR [eax+8], 543649385</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [eax+12], 1948283753</code></td>
<td><code>mov DWORD PTR [eax+12], 1948283753</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [eax+16], 1814065007</code></td>
<td><code>mov DWORD PTR [eax+16], 1814065007</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [eax+20], 560426607</code></td>
<td><code>mov DWORD PTR [eax+20], 560426607</code></td>
</tr>
<tr>
<td><code>mov BYTE PTR [eax+24], 0</code></td>
<td><code>mov DWORD PTR [esp+28], 0</code></td>
</tr>
<tr>
<td><code>mov eax, DWORD PTR [esp+12], 4</code></td>
<td><code>mov DWORD PTR [esp+12], 4</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [esp+8], OFFSET FLAT:LC0</code></td>
<td><code>mov DWORD PTR [esp+8], OFFSET FLAT:LC0</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [esp+4], eax</code></td>
<td><code>mov DWORD PTR [esp+4], OFFSET FLAT:LC0</code></td>
</tr>
<tr>
<td><code>mov DWORD PTR [esp], OFFSET FLAT:LC0</code></td>
<td><code>mov DWORD PTR [esp], OFFSET FLAT:LC0</code></td>
</tr>
<tr>
<td><code>call __printf</code></td>
<td><code>call __printf</code></td>
</tr>
<tr>
<td><code>mov eax, 0</code></td>
<td><code>mov eax, 0</code></td>
</tr>
<tr>
<td><code>leave</code></td>
<td><code>leave</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td><code>ret</code></td>
</tr>
<tr>
<td>LFE12:</td>
<td>LFE12:</td>
</tr>
</tbody>
</table>

Example (4)

<table>
<thead>
<tr>
<th>Vulnerability code in C</th>
<th>Non-Vulnerability code in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;stdlib.h&gt;</td>
<td>#include &lt;stdlib.h&gt;</td>
</tr>
<tr>
<td>#include &lt;string.h&gt;</td>
<td>#include &lt;string.h&gt;</td>
</tr>
<tr>
<td>#include &lt;stdio.h&gt;</td>
<td>#include &lt;stdio.h&gt;</td>
</tr>
<tr>
<td>int main() {</td>
<td>int main() {</td>
</tr>
<tr>
<td>char buf[10], dst[10];</td>
<td>char buf[10], dst[10];</td>
</tr>
<tr>
<td>int n, p;</td>
<td>int n, p;</td>
</tr>
<tr>
<td>fgets (buf, sizeof(buf), stdin);</td>
<td>fgets (buf, sizeof(buf), stdin);</td>
</tr>
<tr>
<td>n = buf[0];</td>
<td>n = 10;</td>
</tr>
<tr>
<td>memcpy (dst, buf, n);</td>
<td>memcpy (dst, buf, n);</td>
</tr>
<tr>
<td>p = 10;</td>
<td>p = 10;</td>
</tr>
<tr>
<td>memcpy (dst, buf, p);</td>
<td>memcpy (dst, buf, p);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
## Appendix D: Prototype of Vulnerability Detection

### Vulnerability code in assembly

```
_main:
LFB12:
push ebp
mov ebp, esp
and esp, -16
sub esp, 48
call __main
mov cx, DWORD PTR __imp___iob
mov DWORD PTR [esp+8], cx
mov DWORD PTR [esp+4], 10
lea cx, [esp+30]
mov DWORD PTR [esp], cx
call _fgets
movzx eax, BYTE PTR [esp+30]
mov DWORD PTR [esp+44], cx
mov eax, DWORD PTR [esp+44]
mov DWORD PTR [esp+8], cx
lea eax, [esp+30]
mov DWORD PTR [esp+4], cx
lea eax, [esp+20]
mov DWORD PTR [esp], cx
call _memcpy
mov DWORD PTR [esp+40], 10
mov eax, DWORD PTR [esp+40]
mov DWORD PTR [esp+8], cx
lea eax, [esp+30]
mov DWORD PTR [esp+4], cx
lea eax, [esp+20]
mov DWORD PTR [esp], cx
call _memcpy
mov DWORD PTR [esp+40], 10
mov eax, DWORD PTR [esp+40]
mov DWORD PTR [esp+8], cx
lea eax, [esp+30]
mov DWORD PTR [esp+4], cx
lea eax, [esp+20]
mov DWORD PTR [esp], cx
call _memcpy
leave
ret
LFE12:
```

### Non-Vulnerability code in assembly

```
_main:
LFB12:
push ebp
mov ebp, esp
and esp, -16
sub esp, 48
call __main
mov eax, DWORD PTR __imp___iob
mov DWORD PTR [esp+8], eax
mov DWORD PTR [esp+4], 10
lea eax, [esp+30]
mov DWORD PTR [esp], eax
call _fgets
mov eax, DWORD PTR [esp+44]
mov DWORD PTR [esp+44]
mov eax, DWORD PTR [esp+44]
mov DWORD PTR [esp+8], eax
lea eax, [esp+30]
mov DWORD PTR [esp+4], eax
lea eax, [esp+20]
mov DWORD PTR [esp], eax
call _memcpy
mov DWORD PTR [esp+40]
mov eax, DWORD PTR [esp+40]
mov DWORD PTR [esp+8], eax
lea eax, [esp+30]
mov DWORD PTR [esp+4], eax
lea eax, [esp+20]
mov DWORD PTR [esp], eax
call _memcpy
leave
ret
LFE12:
```

### Example (5)

#### Vulnerability code in C

```c
main(int argc, char **argv)
{
    char *buffer = (char *)malloc(101);
    int i;
    for (i = 0; i < 10; i++)
        strncat(buffer, argv[i], 10);  // line 7
}
```

#### Non-Vulnerability code in C

```c
main(int argc, char **argv)
{
    char *buffer = (char *)malloc(101);
    int i;
    for (i = 0; i < 10; i++)
        strncat(buffer, argv[i], 10);  // line 7
}```
### Appendix D: Prototype of Vulnerability Detection

#### Vulnerability code in assembly

```
_main:
LFB0:
push ebp
mov ebp, esp
and esp, -16
sub esp, 32
call __main
mov DWORD PTR [esp], 101
__malloc
mov DWORD PTR [esp+24], eax
mov DWORD PTR [esp+28], 0
jmp L2
L3:
mov eax, DWORD PTR [esp+8]
lea edx, [0+eax*4]
mov eax, DWORD PTR [ebp+12]
add eax, edx
mov eax, DWORD PTR [eax]
mov DWORD PTR [esp+4], eax
mov DWORD PTR [esp+24]
mov DWORD PTR [esp], eax
call __strncat
add DWORD PTR [esp+28], 1
jmp L2
cmp DWORD PTR [esp+28], 9
jle L3
leave
ret
LFE0:
```

#### Non-Vulnerability code in assembly

```
_main:
LFB0:
push ebp
mov ebp, esp
and esp, -16
sub esp, 32
call __main
mov DWORD PTR [esp], 101
__malloc
mov DWORD PTR [esp+24], eax
mov DWORD PTR [esp+28], 0
jmp L2
L3:
mov DWORD PTR [esp+8], 10
mov eax, DWORD PTR [ebp+12]
mov DWORD PTR [esp+4], eax
mov DWORD PTR [esp+24]
mov DWORD PTR [esp], eax
call __strncat
add DWORD PTR [esp+28], 1
jmp L2
cmp DWORD PTR [esp+28], 9
jle L3
leave
ret
LFE0:
```
Appendix E Strategies Associated with Vulnerability Detection Model

Table E-1 the Base Strategies Associated with Vulnerability Detection.

<table>
<thead>
<tr>
<th>Goal:</th>
<th>Method:</th>
</tr>
</thead>
<tbody>
<tr>
<td>To test if the software process allows violation of the constraint: data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer.</td>
<td>Attempt to store data larger than the size of the buffer into the fixed length buffer. The constraint is considered violated if the process does not restrict the size of data and copies it into the buffer.</td>
</tr>
<tr>
<td>To test if the software process allows violation of the assumption: the process will not interpret data present on the dynamic memory as executable code.</td>
<td>Attempt to overwrite process variables, such as return addresses and exception pointers. Because these variables are responsible for redirecting the instruction pointer to the appropriate instruction, evaluators can overwrite them to point to an address of their choice. The assumption is considered violated if evaluators are able to redirect the instruction pointer to an address of their choice.</td>
</tr>
<tr>
<td>To test if the software process allows violation of the assumption: environment variables being used by the process have expected format and values.</td>
<td>Restrict the amount of memory available to the software process by running it in a controlled environment using tools such as holodeck. These tools allow control of the amount of memory available to the software process. The assumption is considered violated if the process terminates abnormally or hangs indefinitely.</td>
</tr>
<tr>
<td>To test if the software process allows violation of the assumption: data present on the dynamic memory cannot be observed while the process is in execution.</td>
<td>Execute the software process in a controlled environment, such as a debugger, which allows evaluators to view the contents of the dynamic memory. The assumption is considered violated if evaluators can access any privileged data that the process has stored in the dynamic memory.</td>
</tr>
<tr>
<td>To test if the software process allows violation of the assumption: data owned by the process and stored on the dynamic memory cannot be accessed after the process frees the memory.</td>
<td>Attempt to read the contents of the memory allocated to the software process after it terminates. Since the memory being used by the process is not erased after the process frees it, evaluators can directly access the physical memory and attempt to read data left over by the process.</td>
</tr>
<tr>
<td>To test if the software process allows violation of the assumption: a pointer variable being used by the process references a legal memory location.</td>
<td>Attempt to change the memory location to which the pointer points. The assumption is considered violated if evaluators can modify the value of the pointer variable to refer to memory locations outside the process space or to wrong variables.</td>
</tr>
</tbody>
</table>
Appendix E: Strategies Associated with Vulnerability Detection Model

<table>
<thead>
<tr>
<th>Goal:</th>
<th>To test if the software process allows violation of the assumption: a memory pointer returned by the underlying operating system does not point to zero bytes of memory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>Request zero bytes of memory from the operating system. Some operating systems do return pointers that point to zero bytes of memory. The assumption is considered violated if the process uses this pointer to access memory.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal:</th>
<th>To test if the software process allows violation of the assumption: a pointer variable being used by the process cannot reference itself.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>Attempt to change a pointer to point to it. The assumption is considered violated if evaluators are successful in redirecting the pointer to point to it.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal:</th>
<th>To test if the software process allows violation of the constraint: data accepted by the process must not be interpreted as a format string by the I/O routines.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>Provide the software process with a format string as input. The constraint is considered violated if the process accepts the string and outputs the contents of the program stack.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal:</th>
<th>To test if the software process allows violation of the assumption: the value of an integer variable/expression (signed &amp; unsigned) accepted/calculated by the process cannot be greater (less) than the maximum (minimum) value that can be stored in the integer variable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>Store values that are larger (smaller) than the maximum (minimum) value that can be stored in an integer variable. The assumption is considered violated if the process attempts to store these values, and in doing so, stores overflow (underflow) values, which are different from the intended values.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal:</th>
<th>To test if the software process allows violation of the constraint: an integer variable/expression used by the process as the index to a buffer must only hold values that allow it access to the memory locations assigned to the buffer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>Use index values that are larger than the size of the buffer. The constraint is considered violated if the process uses these index values.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal:</th>
<th>To test if the software process allows violation of the constraint: an integer variable/expression used by the process to indicate length/quantity of any object must not hold negative values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>Use negative values to indicate the length of the objects. The constraint is considered violated if the process uses these negative values.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal:</th>
<th>To test if the software process allows violation of the constraint: data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer on the static memory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>Attempt to store data larger than the size of the buffer into a fixed length buffer. The constraint is considered violated if the process does not restrict the size of the data and copies it into the buffer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal:</th>
<th>To test if the software process allows violation of the assumption: data held on the static memory cannot be observed while the process is in execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>Execute the software process in a controlled environment, such as a debugger, which allows evaluators to view the contents of the static memory. The assumption is considered violated if evaluators can access any privileged data that the process has stored in the static memory.</td>
</tr>
</tbody>
</table>