WEB-DELIVERED ASSEMBLY LANGUAGE INTERACTIVE TRAINING AND ITS SEQUENCE IDENTIFICATION FOR SOFTWARE REVERSE ENGINEERING

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

By

SUNETHRA THIMMAPURAM
B.Tech., Jawaharlal Nehru Technological University, 2012

2017
Wright State University

______________________________
Adam R. Bryant, Ph.D.
Thesis Director

______________________________
Mateen M. Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on Final Examination

______________________________
Adam R. Bryant, Ph.D.

______________________________
Michelle A. Cheatham, Ph.D.

______________________________
Mateen M. Rizki, Ph.D.

______________________________
Barry Milligan, Ph.D.
Interim Dean of the Graduate School
The purpose of this research is to help the beginners of software reverse engineering, the process of extracting knowledge or design information from a product and reproducing it on basis of extracted information. Beginner users often find it difficult to use, as the task may be found daunting at first sight due to a lack of existing learning resources. We need a better way to present information to the novice reverse engineer about how to understand low-level sequences of assembly instructions. This is akin to how the key to learning a foreign language is based upon a fundamental knowledge of the word sequences (grammar).

With this in mind, a web model named ‘WATSRE’ (web-delivered assembly language training for software reverse engineers) is developed to analyze the common patterns of sequences in assembly language. The term assembly language is referred as the low-level programming language for a computer, or microprocessors and other programmable devices. One of the prime reasons to inquire assembly language is that the executable files from the debugger are all in binary instruction format. Moreover, common sequences are identified from the executable files of real world C programs. These sequences identified are translated into meaningful words, which will then be useful for beginners as it is easy to memorize it. Such features developed in this program will assist a beginner in understanding and comprehending these instructions easily. In addition to this, we focus on the working of assembly instructions and their effect on memory and registers. Hence, the finding of this research will be useful in helping the software reverse engineers and people related to this field.
Table of Contents

Abstract.................................................................................................................................................. iii

Acknowledgments .................................................................................................................................. x

1. Introduction......................................................................................................................................... 1
   1.1. Overview........................................................................................................................................ 2
   1.2. Motivation....................................................................................................................................... 3
   1.3. Research Problem......................................................................................................................... 5
   1.4. Approach to Solving the Problem ............................................................................................... 6
   1.5. Thesis Outline ............................................................................................................................. 7

2. Literature Review............................................................................................................................. 8
   2.1. Overview........................................................................................................................................ 8
   2.2. Need for a training system in Software Reverse Engineering ................................................. 8
   2.3. Importance of Assembly language in Software Reverse Engineering .................................... 9
   2.4. Importance of Pattern Mining and Visualization of Text ......................................................... 10
   2.5. Sense Making in Reverse Engineering ....................................................................................... 11
   2.6. Training systems in Reverse Engineering ............................................................................... 14
   2.7. Summary....................................................................................................................................... 15

3. Methodology..................................................................................................................................... 16
3.1. Overview ..................................................................................................................... 16

3.2. Design of Methodology .............................................................................................. 16

3.3. Learning individual assembly instructions .................................................................... 18

3.4. Selecting the concepts ................................................................................................ 18

3.5. Finding the sequences and its translation ..................................................................... 19

3.6. Limiting the instructions .............................................................................................. 23

3.7. Summary ....................................................................................................................... 30

4. Implementation ................................................................................................................. 31

4.1. Overview ....................................................................................................................... 31

4.2. Tools and Technologies used ....................................................................................... 31

4.2.1. Programming Language – AngularJS ...................................................................... 31

4.2.2. Web designing CSS – Bootstrap .............................................................................. 33

4.2.3. IDE to develop - Visual studio code ...................................................................... 34

4.3. Details about the Website ............................................................................................ 34

4.3.1. Home Page ............................................................................................................... 34

4.3.2. Instructions ............................................................................................................... 35

4.3.3. The instruction list block ....................................................................................... 35

4.3.4. Input block .............................................................................................................. 36
List of Figures

Figure 1: Sensemaking Functions in Reverse Engineering [1]....................................................... 13
Figure 2: Methodology of WATSRE model..................................................................................... 17
Figure 3: Block diagram depicting relation of concept, sequences and its translations ............ 22
Figure 4: Step -1 of Instruction Limiting Flowchart........................................................................ 25
Figure 5: Step -2 of Instruction Limiting Flowchart........................................................................ 26
Figure 6: Step- 3 of Instruction Limiting Flowchart........................................................................ 27
Figure 7: Home page.......................................................................................................................... 35
Figure 8: Instructions page................................................................................................................ 35
Figure 9: Instructions page with Label column .................................................................................. 38
Figure 10: Instructions page with an example ................................................................................... 39
Figure 11: Finding patterns page ........................................................................................................ 41
Figure 12: Input text area in finding patterns page .......................................................................... 42
Figure 13: Output area and results in finding patterns page ............................................................... 43
Figure 14: Results area...................................................................................................................... 44
Figure 15: Tabular format of sequences in results area ..................................................................... 44
Figure 16: Help page.......................................................................................................................... 45
Figure 17: Final instruction count comparison among different compilers for Type -1 files ...... 49
Figure 18: Final instruction count comparison among different compilers for Type -2 files ...... 50
Figure 19: Instruction count before and after limiting algorithm .................................................... 51
Figure 20: Comparison of Sequence matching with Godbolt............................................................ 52
Figure 21: Histogram with sequence ID in x-axis and Count on y-axis
List of Tables

Table 1: Relation among Concept ID, Sequence Id, Sequences and its translations .................. 22

Table 2: Example of Hello world program showing addresses and instructions ..................... 28

Table 3: Reduction of instructions in Hello world program example .................................. 29

Table 4: Check for model validation and verification .......................................................... 48

Table 5: Standard deviation calculation for Type -1 .......................................................... 49

Table 6: Standard deviation calculation for Type -2 .......................................................... 50
ACKNOWLEDGMENTS

Many people have pledged support to ensure the success of this project. I would like to take this opportunity to thank my friends, colleagues, and everyone else who has been with me every step of the way. I would like to express my sincere appreciation for those who have provided me with so many chances to complete my thesis.

Thank you, Mom, Dad, and Sister, for your love and the unconditional support you have given me. I would also like to pay my respect and immense gratitude to my professor, Dr. Bryant, who has guided me throughout the thesis with patience and intelligence. A very special thanks to Dr. Cheatham and Dr. Rizki for their invaluable support in developing this thesis.
1. Introduction

Despite the growing popularity of high-level languages such as Java, Visual Basic.NET and C-Sharp, assembly language is still significant in the contemporary context because it is a low-level programming language and its instructions are immediately translated to the machine language. It enables the developers to control different aspects of the hardware and the programmers to highlight critical issues by writing special processor-based instructions. Some of the useful code-blocks that can be written in assembly language include device drivers, boot codes, and embedded systems. The main benefit of writing code in assembly language is that the programmers get the closest access to the machine’s processor and also access the registers and memory addresses by writing instructions in it. Moreover, it provides significant improvements in speed and response time.

Recovery of the design is the most dynamic subdivision of reverse engineering. Even though dissimilar high-level languages are being used in the application development, the importance of assembly language with reverse engineering has its own wonders [2]. The main focus of this research is to incorporate assembly language to help beginners in reverse engineering find a platform, and to apply a web-based opcode model for binary analysis training. Because the recovery of the design must make a replica of all the information compulsory for someone to fully understand the nature of the program, it deals with a broader perspective of information and statistics than what is found in the ordinary software-engineering code of representation.
1.1. Overview

The assembly language is closest to the mainframe computer machine. This thesis develops an interactive model for learning assembly instructions and memorizing the assembly patterns provided from a decompiled C program, which will help beginners in software reverse engineering. In general, reverse engineering is a technique where a system is dismantled in order to study its architecture and blueprint, ranging from electronic devices, machinery components, and the like [7]. This technique is quite commonly performed in various walks of life, both individually, as well as at an industrial level. In the case of computing, similarly, when the binary code of the software is used to generate its source-code, it accounts for “software reverse engineering”.

The model developed in this thesis can be used as a self-teaching tool by students or any interested individuals who lack the otherwise required proficiency in assembly programming. This tutorial is therefore versatile as it can also be used by instructors of Reverse Engineering or Assembly Learning courses (such CSE101) to assist their classroom teaching. SRE, (Software Reverse Engineering) is the practice of analyzing a software to implement information and extraction of the design. A classic SRE situation is a software syllabus that carries numerous business rules in its code lines is misplaced; what is left is "binary" or "native" code. The skills of reverse engineering are utilized to distinguish and deactivate malware and viruses and protect intellectual property. It became alarmingly obvious in the period of crisis of Y2K that these skills were generally practiced among the programmers. Since, extensive research has been made in concern to determine the types of activities that fall under reverse engineering for the sake of teaching these skills to the tester and the programmers.
This model is given a user-friendly interface which will guide the user(s) through the working of individual and/or assembly instructions in 32-bit and 64-bit and demonstrate their effect on registers and memory. In addition, analyzing and memorizing common patterns from a group of assembly instructions, which can be obtained from the decompiled real C program (the assembly instructions were obtained by limiting the total count of instructions from an input exe which is in text file format). All this is presented along with references to several works in the literature.

1.2. Motivation

The key motivation and the fundamental element for this thesis are rooted in Dr. Bryant's on-going work, which is based on a proposal to add a KIL (Knowledge Interactive Levels) training plugin to an existing reverse engineering tool [8]. The main aim of Dr. Bryant’s proposal is to provide a training/learning system for the beginners in reverse engineering. Although I have not adopted his entire idea of creating a plug-in, I have taken his objective of developing a model, which can be useful for reverse engineers. In this process, the research will focus on creating a web-based opcode model, for binary analysis training of beginners in software reverse engineering.

Cybersecurity has indisputably become a subject of utmost importance in all lifestyles, but, more so with technological advancements such as the Internet of Things (IoT) and cloud computing, leading to ubiquitous digitalization. Just as humans require protection of police services and physical locks, organizations and government sectors demanded an information security system, that plays an essential role in controlling the organization’s security and privacy risks [11]. According to US Department of Justice, ransomware attacks quadrupled to 4,000 per
day from 2015 to 2016. The Global Risks 2015 report [9], published by the World Economic Forum (WEF), included a rather stark warning: "90 percent of companies worldwide recognize they are insufficiently prepared to protect themselves against cyber-attacks" [9].

In today's world, the threat to cybersecurity is increasing at an alarming rate. Both private and governmental networks are targeted. In order to protect one’s system from breaches, private and public sectors are seeking for solutions to stop the ones responsible for the attacks. While most people would agree organizations have the right to safeguard their systems from cyber-attacks; however, they do not have a clear idea about what options are included in the legal and moral defense of the network [12]. The passive defense might be a solution within the realm of options open to the defender. A ‘hacked system' can mean anything from stolen data to cracked passwords, leading to the access of several associated applications, services, networks etc., using illegal authorization techniques and paving the way to several institutional (or individual) security breaches. Hence, the need to constantly discover defense strategies to surmount the perpetual challenge of digital information theft is very important.

One of the key applications of software reverse engineering is its use as a testing mechanism to detect major vulnerabilities and security flaws in a system. For instance, this technique can be designed to assist its users in learning how the attacker entered the system and how the breach was discovered, following a security compromise [7]. It is well-known that all computing systems have binary code included as the low-level programming language. Using reverse engineering debuggers, security analysts can trace the changes made following the breach, thereby identifying its root cause. It is to be understood by the users, however, that once such techniques of reverse engineering are learnt, it is mandatory upon them to use it responsibly and solely for legal purposes, and that any illegal usage is strictly prohibited and ensures legal
actions from federal departments such as the Department of Homeland Security (DHS), FBI, etc [7].

Hence, in order to train a newcomer in software reversing, one of the most important things is learning assembly language due to the role of binary analysis and decoding in Reverse engineering. Debugged code is in assembly format, which is a binary, low-level programming language that plays an important role in software reverse engineering..

This proposition specifically discusses developing a web model that facilitates its users’ rapid learning of the fundamentals of assembly language and thus software reverse engineering.

1.3. Research Problem

Training many institution personnel to stay ahead of the game in cybersecurity is the key to its very survival in this age of information vulnerability. The versatility of software reverse engineering is evident in its wide-ranging applications, such as malware detection, cryptographic algorithm debugging, digital rights management, software enhancement through reversal of existing model, and software quality evaluation.

Training novice reverse engineers is time-intensive and therefore, it is quite expensive. The current market lacks sufficient supply to meet the growing demand for software tools that can teach assembly techniques to beginners. Although there are plenty of written reports and books in the literature which discuss the nuances of this subject, the need for an interactive learning tool at a user's disposal cannot be overstated, considering the bountiful applications software reverse engineering offers. Such a tool, implemented as an interactive webpage or executable software, integrates adaptive learning techniques thereby providing the users with real-time implementation of the techniques involved, thereby complementing the user's
theoretical knowledge, gained from the works in literature.

Therefore, the goal is to improve the reverse engineering training through pattern mining of important instructions and memorizing them. In order to do this, we do not know which patterns are useful, so this has to be investigated. After identifying which are important patterns, an algorithm or a method to extract these patterns from a stack of assembly instructions has to be developed. Once these patterns are found, they have to be translated into higher-level representation for the user to understand their meaning.

1.4. Approach to Solving the Problem

The name of this work, “WATSRE - Web-delivered Assembly language training for Software Reverse engineering”, proposes the fact that it falls under the broad domain of “Training in Cyber”. A web-based model is proposed, which not only allows the user to apply the theoretical knowledge gained from classroom learning or literature reading but also facilitates self-assessment throughout the process at each step. Reverse engineering is a kind of certification, which makes it mandatory to recognize the security experts and prove knowledge in the core areas of aggressive cyber threat discovery and justification. To justify and play as an assistance, this model has been designed to be helpful for beginners to use.

This model is designed to be interactive and easy to use. At this point, the developed model consists of different tabs for the use of different functionalities like learning the operation of assembly instructions, limiting the instruction count and identifying the patterns. The following are the key challenges addressed in this work:

- How to identify the patterns directly from the assembly languages of real programs?
- What information is important to be kept in or left out of an instruction pattern?
- Are the actual registers and addresses important to one of these pattern templates?
- Do only sequence patterns matter, or do we also care about patterns that are not in a strict sequence?
- How does the limiting of instructions affect the pattern identification?

The answers to these questions are discussed in detail in Chapter 3 Methodology.

1.5. Thesis Outline

Chapter 1 initially talks about overview and motivation of this thesis topic, and later, the problem statement and approach to the solution. Chapter 2 goes over the literature review of reverse engineering tutorial systems, assembly training programs, and the importance of pattern recognition. Chapter 3 explains the methodology of the WATSRE model, by justifying the approach to the problem statement. In Chapter 4, the implementation of the tutorial is presented by discussing in detail the nuances of each module and its working. Finally, in Chapter 5, the results, conclusion and future prospects are discussed.
2. Literature Review

2.1. Overview

This chapter discusses in detail the contributions of several reports in literature that helped in making this WATSRE model. It is classified into three sections: first, the need for a training system in software reverse engineering is discussed in section 2.2. Secondly, the importance of assembly language in reverse engineering is explained in section 2.3. Then, in section 2.4, the importance of pattern mining and visualization of text is discussed. Sense making and visualization in understanding the reverse engineering techniques is, discussed in section 2.5. Later, the current tutoring systems are discussed in section 2.6 and finally section 2.7 concludes the literature review.

2.2. Need for a training system in Software Reverse Engineering

The need for a learning system can be justified because, Smith, et.al., [10], for instance, discussed the challenges of teaching reverse-code engineering in a classroom setting. Resource and technological growth of other computer science disciplines were compared, in their work, to reverse engineering, concluding that the latter had smaller growth. The team attributes the reason for this lag to legal difficulty, limited background, a lack of codification of the existing breadth of knowledge, and most importantly, a lack of qualified workforce for testing [10]. They proposed a framework for artificial-intelligence-based expert systems (ES) for reverse engineering tasks, their goal being to transfer expert knowledge in a domain to a system. In
general, expert systems have three parts: knowledge base, working memory and inference engine [12]. They discussed multiple forms of expert systems, including intelligent tutoring systems (ITS).

They proposed for ITS to be packed in modules such as student or tutor, in addition to a user interface, which can be type-rule-based or knowledge-based systems. Based on the requirements, more modules can be added in order to assist the user. The tutoring model being discussed is adapted from “Behavior of Tutoring systems” by Vanlehn [13].

Eventually, the team concluded by proposing for future work in this field to focus on formalizing and integrating the knowledge of reverse engineering. This thesis thus justifies the need for a learning system for reverse engineering research, focusing on building its foundations in order to apply the same ideas to fighting cyber security threats.

2.3. Importance of Assembly language in Software Reverse Engineering

Reverse engineering is defined as the process of deriving data from design and knowledge of a product and reproducing it [29]. Assembly language, also known as ASM, is considered the lowest level of programming language that is easily understood by people. It is a machine language that can be easily handled and enable people to use a computer properly. In addition, it allows to code using different program languages. Programs coded in assembly are small and can easily communicate at a faster rate with the machine. Therefore, assembly language plays a crucial role in reverse engineering [28]. Primarily because each assembly instruction is linked with a code that remains the same throughout, therefore it uses a mnemonic device to represent each low-level machine opcode (operation code). This, in return, helps in
reverse engineering.

Other than this, assembly language is applicable to different processes due to its function of decompilation, especially when reversing a MIPS executable file. Whereas, without assembly language it is not possible to decompile an executable file into a high-level language (Eilam, 2005). In addition to this, reverse engineering can serve as an important tool for acquiring knowledge on how to use a specific programming language or how to work on any specific type of application. Therefore, it is imperative to have basic knowledge about the language in which any program is written [27].

2.4. Importance of Pattern Mining and Visualization of Text

The term text mining refers to the process of acquiring high-quality information from a collection of texts. It is a facilitating technology with applicability across learning, research, and management [18]. The key advantage of text mining is that it structures the input text, extracts patterns within the structured data, and conducts analysis by interpreting the data [26]. Apart from this, it is beneficial to businesses by identifying valuable information from text-based content, for instance, social media and CRM systems. Specifically a tool such as the Digital Editing Professional can assist them in mapping, visualization, and editing [19].

Text mining also offers solutions to many problems by drawing on techniques from information recovery, data mining, and natural language processing. Another benefit of using text mining is that it allows quick analysis of language and word usage. It also helps to make inconceivable scale readable by making sense of the information quantitatively. Through this, professionals are able to compare the vocabulary of different documents, trace the history of the language from the past and most importantly can categorize them[19].
Moreover, various tools of text mining play a vital role in scientific fields, especially fields related to education, as it allows researchers to find and assemble various research papers into one place. Student performance in different courses can be assessed and how different characteristics affect the selection of subjects can be analyzed [30].

In this thesis, text-mining concepts play an important role in finding the sequences of instructions from the disassembled file.

2.5. Sense Making in Reverse Engineering

The term ‘Sense-making’, in information science, first coined in the work of Dervin et.al., denotes the ability to provide an understanding from gathered data and for decision-making purposes [4]. Zhang and Soergel in their work [5] provide an overview of how the sense-making process supports intellectual techniques and the tools that assist the same. A sequential process for making sense of any given task, computational or otherwise, is laid out in their work, after a review of available sense-making theories and models.

Dr. Bryant in his dissertation [6] detailed a theory on how executable programs make sense to a reverse engineer. A case study was performed to examine how reverse-engineering tasks are performed by individuals, with focus specifically on the sense-making part of the task. The resulting theory would help enhance the tools used in reverse-engineering by augmenting the robust computational models in complex tasks, improving the reverse engineering training needs, and identifying the vulnerabilities and malicious content present in the executables.

Bryant, Mills, Peterson and Grimaila synthesized and refined the sense-making and situation-awareness models with cognitive process in software reverse engineering[1]. They start the discussion with the different methods of data representation in the reverse engineering
environment, and mention that since the source code of the program is unavailable, the executable form should be analyzed, which is in assembly language[1]. Understanding the assembly code requires an extensive insight of high level programming knowledge, owed to its complicated structure, compared to coding language. As it is well known, even a relatively simple program written in conventional coding language contains several thousands of instructions when translated to assembly language. A few other ways of representing data include program, visual and instrumentation. In program data, instructions are represented by strings of bytes, whereas in visual data, each block, divided by jmp or call instructions, are connected and visualized as graphs, and instrumentation data, which is from system probes, provides information in systems running in real-time. In-built third-party tools can be used to detect this form of data and it acts as an additional information source to the reverse engineer [1]. It can get increasingly difficult for a single disassembler/debugger to showcase several of these data formats for an executable file, under a single roof. Hence, only those debugger and disassembler combinations are chosen, that are suitable for the desired data format in a reverse-engineering process.

In order to explain the sense-making process, Bryant et. al., in this work, use the models prepared by Klein and Zhang. The performance of the existing knowledge schemas - procedural and declarative- are compared using examples. It is theorized that the procedural knowledge of a reverse engineer depends on the interaction of the organized patterns stored in their mind. Their declarative knowledge depends on the information they have such as, taxonomy and the application of their procedural experience, which experts would possess more of, compared to novices. Another important knowledge type is domain-specific, which novices of a given field need to acquire in order to perform reverse-engineering tasks. Once this is achieved, the user can
then develop procedural and declarative knowledge in that domain.

They designed and explained the sense making functions in a hierarchical manner, the steps involved being: 1) goal construction 2) planning 3) carrying out the plan 4) generating hypotheses or questions 5) determining needed information 6) experimentation to seek data 7) instrumentation to isolate unavailable data 8) evaluating and integrating 9) updating the mental model [1]. All these functions are shown in figure 1.

![Figure 1: Sensemaking Functions in Reverse Engineering](image)

Bryant et. al. explained the functions as follows: the first and foremost step is to set a goal, then comes planning, which requires knowledge of the problem statement, skill, effort and time. During this phase, if an idea seems to work well, they carry out the plan. As seen in the figure 1, if the set goal is achieved after carrying out plan ‘A’, then the problem is solved; but, if plan ‘A’ does not work, the flow chart dictates to go to the generate question step where the reverse engineer is supposed to develop a hypothesis. During the hypothesis, the user formulates
a set of questions which can hint to different causes of failure. The approach, then, is to determine the missing information. For this to happen, output data is collected from the instrument system. The loop to instrument system keeps running until the required data is obtained. As soon as the information is collected, the system evaluates the new knowledge and checks if it has to be updated to the existing structure or not. If the reverse engineer interprets that the new knowledge is valid, it will be updated in the mental model. If the new knowledge is not valid, then the user should return to the generate hypothesis step and repeat the process [1].

At this point, it can be inferred that, in software reverse engineering, a trained engineer is required to understand the functionality of complicated programs for which source code is not available [6]. This highlights the importance of the sense-making process in reverse engineering.

2.6. Training systems in Reverse Engineering

Currently, with the increase in the use of technology, the impulse for gaining knowledge has also increased. Due to timing or financial constraints, it may not be possible for all interested candidates to take a formal education and attend school. Therefore, in recent years, there has been a huge demand in the development and usage of web based and software based tutoring systems.

Brusilovsky, from Carnegie Mellon University, has done research on web-based education and explained its importance [15]. He mentioned that the number of users utilizing self-teaching education applications has been drastically increasing. He also explained the current problem of such applications or websites being simple hypertext (html) pages and consequently provided a research goal to advance these applications[15]. So, the need for adaptive, interactive systems arose, which aims at providing knowledge to all the users.
When looked up for an online training tool for reverse engineering, most of them turned out to be either online courses or videos or blogs or theory that teaches bits and pieces of binary. This is the same with the case of assembly instructions; to my knowledge, no interactive systems are present to teach these techniques. Moreover, the courses provided to train in these technologies are not free. ‘Tuts for you’ by Lena is the notable reverse engineering tutoring systems for beginners. This site consist of binary executables with instructions provided to work on crackme’s and it has received mixed reviews from the users. Hence, there is a strong need to develop online training systems in reverse engineering.

2.7. Summary

This literature review gave an overview of the different works of reverse engineering, sense making and web-based learning systems. To fill the knowledge gaps in the aforementioned researches, a tutorial system in reverse engineering has to be designed. With the background and foundation from the above sources, a methodology has been designed and will be discussed in chapter 3.
3. **Methodology**

3.1. **Overview**

To develop the software reverse engineer, it is necessary to learn about the assembly instructions and to memorize the patterns. This work has been divided into two parts: the first is to learn the instructions and the second is to find the patterns. The former step is to make the user familiar with the understanding of assembly instructions prior to working with the patterns and the later one is to find the sequences. In order to perform both the tasks, a model must be developed to help the user visualize and learn the instructions flow in the registers and memory. The second step is to make this model to limit the number of instructions required, identify the sequences, and help the user to memorize the same.

3.2. **Design of Methodology**

Once the research topic and techniques are determined, there are certain elements that need to be finalized. These include selecting a debugger for analyzing the assembly, choosing the concepts for which the sequences have to be found, analyzing the sequences, and ensuring that they transform to an appropriate meaning.

First is, selecting a debugger: there are many de-compilers and debuggers available over the web. However, x32dbg/x64dbg is selected due to its easy usage and excellent graphical user interface (GUI). The user interface is loaded with different colors in order to distinguish different instructions and also highlight some of them. The way it works is similar to ollydbg but it seems that x32dbg is more user-friendly and efficient. In addition to this, as compared to other
debuggers, it is easy for a beginner in reverse engineer to install, understand and work on it. Therefore, x32dbg is chosen considering its convenient usage.

"Microsoft Visual Studio Express 2010" compiler is chosen for compiling the C code. The .exe file generated from other compilers like GNU GCC and Borland C++ were compared with the file generated using Visual studio express and it was found that the later one was easier to use. Though they all share the similar types of instructions and similar that counts, the order in which the instructions are present matters in this research. It is easier if the order of assembly instructions is the same as the order of c program, and this feature is observed using the Microsoft visual studio express compiler, hence it is chosen.

Below is the flow diagram that represents the top-down approach for building the WATSRE model. As seen in figure 2, the research methodology is divided into two main branches, one for training the instructions and the other for sequence identification.

Figure 2: Methodology of WATSRE model
3.3. Learning individual assembly instructions

Assembly language is programming done on a low level that is accessible for any mainframe workstation. The code, is executed in a straight line on the CPU, using instructions and numbers. It is very much resembles codes of machine instructions and is able to make hardware and enactment administration possible. Moreover, it is responsible for accessing the superior computer instructions [16].

This section deals with the right branch mentioned in figure 2. When the hardware of the machine has a 32-bit processor, the arithmetic logic unit, data bus, and registers all have a length of 32-bits. Hence, these processors can only accept 32-bit instructions. On the other hand, 64-bit processors and operating systems can handle instructions of 64-bit length. In this thesis, the working of assembly instructions and their effect on registers is explained for 32-bit and 64-bit processors. The instruction set of the assembly language is classified into five categories. These include concepts covering arithmetic, logical program flow, stack management, data movement, and functions. The most basic and important instructions, a total of 24, are chosen from all the categories and their working is demonstrated. More details about each instruction is explained in ‘The Instruction List block’ in section 4.3.2.

3.4. Selecting the concepts

The first step is to identify the patterns. However, it is imperative to opt for the right pattern so I have decided to run the real-world C programs, get the executables of them and observe how each C program is converted to the binary format in the debugger. Now that it has been decided to work on the C programs, the second most important thing is to select the concepts within them in order to identify the patterns. Therefore, the basic and fundamental
topics in C programming are chosen. The chosen topics include file start, file end, variable assignments, read and write operations, arithmetic operations, relational operators and looping. Currently, the system is limited to the topics mentioned above.

**3.5. Finding the sequences and its translation**

After selecting the concepts, the C programs are formulated and collected from various sources in accordance with this. These programs are compiled in the code-blocks IDE using the Microsoft Visual Studio Express Compiler. When the C program compiles and executes successfully, the executed file ‘.exe’ is saved and opened in the x32dbg. After opening and running the .exe file in the x32 debugger, the decompiled C code with memory addresses, registers, opcodes, etc., are displayed.

Our primary focus here is to observe if there are any repeated patterns in the code and results showed there were. A text analyzer from Online-utility.org is used for detecting the common patterns and this tool helped in observing the patterns. The patterns found were just random and did not translate to anything or make sense; but, it is better to check if some pattern format exists in the assembly.

Now that we know that there are patterns in the debugged file, we’ll have to find out the meaning of them. However, the problem now is to find which group of instructions translates to the actual meaning in the real C code. To solve this, test data programs are collected and are compared among themselves for any similar patterns occurring across them. If found, it is checked against its actual C file and the meaning of that pattern with respect to it is investigated. This verification is done on all the C programs that were selected under an individual concept. Moreover, the internet is used as a medium to search various techniques to find the meaning of
different patterns. In order to cross check or validate the sequence and its translation, the Godbolt compiler and x-86 disassembly wikibook are used.

Godbolt is an online compiler which converts C programming to assembly language. Although there are many online compilers, this one has an interesting feature, which is to highlight the set of assembly instructions corresponding to its C code. This compiler website is divided into two windows, the first window, where the C code is given as input and the second window, to display the compiled assembly. For example consider a simple C program as shown below.

```c
#include <stdio.h>
int main()
{
    printf("Hello Reverse Engineer");
}
```

On hovering the open brace ‘{‘ in the first window, the assembly instructions `push rbp, mov rbp, rsp` are highlighted in the second window and it works vice-versa. To validate a sequence which is identified as a part of this thesis, the C program that contains the corresponding concept of that sequence is given as an input to Godbolt to verify if it is correct or not. Since Godbolt uses a different compiler (GCC) and is of 64-bit type, all the sequences identified are not exactly same; there are slight differences in the names of instructions and registers.

In addition to this, x86 disassembler wiki book is also used as a reference manual to understand the meaning of various assembly instructions. It covers concepts like variables, stack, floating point numbers, branches, calling conventions and etc., (Topics referred in this thesis are only mentioned) [32]. This book does not mention about the sequences or list them, but explains the meaning of a few frequent and important instructions when grouped. It provides an overview
of assemblers, compilers, platforms and tools used for reverse engineering, code patterns, data patterns and difficulties of x86-disassembly. This answers the first investigative question on how to identify the patterns from assembly.

In this thesis, while identifying patterns/sequences only instructions are selected, the other elements like the addresses, opcodes and the references are not taken into consideration. Since the assembly language plays a vital role in understanding the flow of instructions, this research only focusses on it. Moreover it is comprised of English words, which helps a person to read and learn quickly when compared to the opcodes which are just numbers. This justifies the answer for second investigative question asked about which information to be kept or left out of an instruction pattern. The importance of assembly is explained in detail in section 2.3.

Now that we can find the sequences and their meanings, we have to make a logical relation between concepts and the sequences. This is done by listing all the concepts and each concept is given an ID, then C programs are executed for each concept with different inputs and observe how the patterns vary. In case the behavior is similar, then consider and list that as a sequence with an ID. This indicates that the concepts are like parent class and the sequences mapped to it are child class. However, one concept can have multiple sequences. An example of a concept and sequence are shown in the table 1. The block diagram representing this format is shown in the figure 3.
Figure 3: Block diagram depicting relation of concept, sequences and its translations

In the table, the first column indicates the concept, the second column indicates the sequence ID, the third column indicates the sequences and the final column has the meaning of the sequences. In this example, the concept is about declaring variables, and its sequences are about how to declare integer, float, character and double values.

<table>
<thead>
<tr>
<th>Concept ID</th>
<th>Seq ID</th>
<th>Sequences</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concept -3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declaring Variables</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>push &lt;reg&gt;</td>
<td>Start of the function; Variables are declared and memory is allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov &lt;reg&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sub &lt;reg&gt;,#</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>mov dword ptr ss:[&lt;reg&gt;-#],#</td>
<td>Integer type variable is moved into the register</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>fstp dword ptr ss:[&lt;reg&gt;-#]</td>
<td>Float type variable is moved into the register</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>fstp qword ptr ss:[&lt;reg&gt;-#]</td>
<td>Double variable is moved into the register</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>mov byte ptr ss:[&lt;reg&gt;-#],#</td>
<td>A Character is moved into the register</td>
</tr>
</tbody>
</table>

Table 1: Relation among Concept ld, Sequence ld, Sequences and its translations
Finally, by this process, a total of 45 sequences are identified under 7 concepts. This number can be expanded in the future. In this thesis, the developed model only identifies the sequences that are in a strict order. But, we do care about the sequences which are not; concepts like while, do-while, if-else and for loops follow a pattern in converting corresponding C to assembly; but, there is a problem here. All the C programs using these loops do not have the same number of variables, same type of expressions, print statements, scan statements etc., So with these additional things, the instructions might increase or decrease across various programs; Thus it results in the change of the instruction order and becomes unpredictable to suggest the strict sequences in the loops. Though these are not implemented in this model, they are provided along with the total list of sequences and the student/user can refer to this list to learn the behavior of these loops. Thus answering the research question on the strict order of sequences.

3.6. Limiting the instructions

Now that we have sequences and concepts, if we write an algorithm for all these sequences, program it and apply for any of the .exe files, imagine what would happen? Yes, it would definitly find the patterns from it; but it’s going to be huge in number because of the high input instructions data. With such a vast number of sequences, a novice reverser might find it difficult to interpret the assembly to C programming. As a beginner, it would be confusing, time consuming and hard to focus on all of them. In this case, the chances are very likely that most of the sequences are going to be repeated a large number of times. In order to overcome all these cons, a method which can extract/limit only the required lines of the assembly that represents the C program has to be developed. After observing the process in numerous executables and comparing with some similar real-time systems, an algorithm is designed to perform this task.
This algorithm will be able to deliver the set of assembly instructions in an order that represents the program flow in C language and this extraction is possible with the C files compiled using Microsoft visual studio express compiler.

This algorithm is validated using the two approaches mentioned by Dr. Hillston in his model validation techniques lecture[31]. One approach is to compare this with the existing systems and in order to do that, the Godbolt and RetDec compilers are used. Though these compilers do not perform exactly like the developed model, certain things have been compared using them. These includes instruction count, percentage similarity of the output instructions, the matching percentage of addresses, and opcodes. The second approach is make a comparison within the existing system using multiple input files. C program files with selected concepts are used for verifying this. The reduced instruction count matched with a precision of +/- 10 instructions for the C programs with no ‘functions’ concept. Another observation is when the C files with ‘functions’ concept are executed, the instruction count vary +/-150. This answers the last investigative question on the affect of limiting instructions in finding sequences.

The instruction limiting is done in three steps and is represented in the flow chart shown below. Step-1 of the flow chart aims to find the first return instruction, step-2 will aim to find the highest call address value, and finally, Step-3 will separate the required instructions from the total.

In step-1, the total instruction count is calculated and its value is stored in a variable ‘n’. Then, search for the first ‘ret’ instruction from these ‘n’ lines, and if found, store the address of it in a variable ‘l’. If there is no ‘ret’ instruction, then the program stops.
Figure 4: Step -1 of Instruction Limiting Flowchart
After retaining the line number in the variable ‘i’, all the call instructions before i\textsuperscript{th} line are looked up. From this the highest address call by any ‘call’ instruction is found and is stored in a variable ‘High’. If there is no ‘call’ instruction in the program, then the i\textsuperscript{th} line becomes the last line of the program and all the instructions after that are eliminated.

Figure 5: Step -2 of Instruction Limiting Flowchart
In step-3, the instruction present in the highest addresses is checked and if it is ‘push c’ and the above one is ‘ret’, the instructions present below push c are removed.

Figure 6: Step-3 of Instruction Limiting Flowchart
This procedure of instruction limiting is explained with an example shown below.

<table>
<thead>
<tr>
<th>Lines</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>010D1000</td>
<td>push ebp</td>
</tr>
<tr>
<td>2</td>
<td>010D1001</td>
<td>mov ebp,esp</td>
</tr>
<tr>
<td>3</td>
<td>010D1003</td>
<td>push hello.10DB000</td>
</tr>
<tr>
<td>4</td>
<td>010D1008</td>
<td>call hello.10D1014</td>
</tr>
<tr>
<td>5</td>
<td>010D100D</td>
<td>add esp,4</td>
</tr>
<tr>
<td>6</td>
<td>010D1010</td>
<td>xor eax,eax</td>
</tr>
<tr>
<td>7</td>
<td>010D1012</td>
<td>pop ebp</td>
</tr>
<tr>
<td>8</td>
<td>010D1013</td>
<td>Ret</td>
</tr>
<tr>
<td>9</td>
<td>010D1014</td>
<td>push C</td>
</tr>
<tr>
<td>10</td>
<td>010D1016</td>
<td>push hello.10D99F0</td>
</tr>
<tr>
<td>11</td>
<td>010D101B</td>
<td>call hello.10D23F0</td>
</tr>
<tr>
<td>12</td>
<td>010D1020</td>
<td>xor eax,eax</td>
</tr>
<tr>
<td>13</td>
<td>01287DB9</td>
<td>add byte ptr ds:[eax],al</td>
</tr>
<tr>
<td>14</td>
<td>01287DBB</td>
<td>add byte ptr ds:[eax],al</td>
</tr>
<tr>
<td>15</td>
<td>01287DBD</td>
<td>add byte ptr ds:[eax],al</td>
</tr>
<tr>
<td>16</td>
<td>01287DBF</td>
<td>add byte ptr ds:[eax],al</td>
</tr>
<tr>
<td>17</td>
<td>01287DC1</td>
<td>nop</td>
</tr>
<tr>
<td>18</td>
<td>01287DC3</td>
<td>nop</td>
</tr>
<tr>
<td>19</td>
<td>01287DC5</td>
<td>nop</td>
</tr>
<tr>
<td>20</td>
<td>01287DC7</td>
<td>nop</td>
</tr>
<tr>
<td>21</td>
<td>01287DC9</td>
<td>nop</td>
</tr>
<tr>
<td>22</td>
<td>010D1020</td>
<td>ret</td>
</tr>
<tr>
<td>23</td>
<td>010D1022</td>
<td>push C</td>
</tr>
</tbody>
</table>

Table 2: Example of Hello world program showing addresses and instructions
In this example, the total number of instructions is 23. In the actual disassembled file, there will be 1000’s of instruction lines; since it is a larger number and to explain the concept with an example it is shortlisted to 23. According to the flowchart, step-1 is to check for the first ‘ret’ instruction. In this example, the first ‘ret’ instruction is present in the line 8. Now, check all the addresses that start with ‘call’ instruction before the first ‘ret’. Make sure that the addresses in that instruction column are present in the address column. Now find the highest address in the ‘call’ instruction and before the first ‘ret’. In the example shown, we have ‘call hello.10D1014’ (line 4) before the first ‘ret’ instruction (line 8) and the highest address is present in 10D1014 (line 9); Once it is found, go to that highest address line and check if that instruction is ‘Push c' and it's above instruction is ‘ret'. If this condition is satisfied, then eliminate all the instructions below ‘push c'; so if we navigate to the address 10D1014 at line 9, the instruction is ‘push c’ and its above instruction is ‘ret’. The abstracted instructions are displayed as shown below:

<table>
<thead>
<tr>
<th>Lines</th>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>010D1000</td>
<td>push ebp</td>
</tr>
<tr>
<td>2</td>
<td>010D1001</td>
<td>mov ebp,esp</td>
</tr>
<tr>
<td>3</td>
<td>010D1003</td>
<td>push hello.10DB000</td>
</tr>
<tr>
<td>4</td>
<td>010D1008</td>
<td>call hello.10D1014</td>
</tr>
<tr>
<td>5</td>
<td>010D100D</td>
<td>add esp,4</td>
</tr>
<tr>
<td>6</td>
<td>010D1010</td>
<td>xor eax,eax</td>
</tr>
<tr>
<td>7</td>
<td>010D1012</td>
<td>pop ebp</td>
</tr>
<tr>
<td>8</td>
<td>010D1013</td>
<td>ret</td>
</tr>
<tr>
<td>9</td>
<td>010D1014</td>
<td>push C</td>
</tr>
</tbody>
</table>

Table 3: Reduction of instructions in Hello world program example
The total instruction count will also be reduced to 9 and displayed in the results window. The above-extracted instructions will be executed through a multiple if-else loop and are translated to a readable format. If an instruction does not fall under any of these conditions, no sequence will be identified from it.

3.7. Summary

The primary aim of this methodology is to come up with a strategy on how to develop different strategies for finding patterns and learning assembly instructions. For this purpose, a design methodology is developed and each block in it is explained in detail. Microsoft Visual Studio Express compiler is used for compiling the C code and its assembly is debugged in x32dbg. Once the concepts are selected, the C programs are formulated and collected from different sources. After compiling and executing these C programs, the .exe file is run in the x32 debugger which displays the disassembled C code with memory addresses, registers, opcodes etc. Patterns of sequences are identified and translations are made. Later, the research focused on limiting the instructions. Moreover, two techniques from model validation and verification [31] are carried out to do the evaluation. In this chapter, the answers to the investigative questions are provided.
4. Implementation

4.1. Overview

The procedure of developing the web-model based on the methodology which is discussed earlier is explained in this chapter. The tools and technologies used for the creating the WATSRE model are discussed in section 4.2 and in section 4.3, the development and working of every page of the website is explained. Screen shots are provided to visualize and describe the different elements on each page.

4.2. Tools and Technologies used

4.2.1. Programming Language – AngularJS

AngularJS is an open source framework written in JavaScript that is used for developing the front-end of web-applications. It was developed by a team at Google to overcome the challenges faced in building single page applications (referred to as SPA’s). It was initially released in October 2010 and the latest stable release was introduced in August 2017. When compared to the other prominent frameworks like React.js and Ember.js, it is the oldest and has a wide user base [20], [21].

This framework aims to provide two different types of architectures:
1) Model-View-Controller (MVC) architecture used in Angular 1 for client side
2) Model-View-ViewModel (MVVM) architecture used in Angular 2

MVC separates an application into three parts (Model, View, Controller) that are interconnected. The model is responsible for managing the data and behavior of the application
providing information about the current state, and responding to instructions for the change of state. The view is responsible for the front end and displaying of the information. The controller is responsible for interpreting the inputs of the users and updating the model, or the view, to take appropriate actions.

The MVVM model emphasizes differentiating the layout from the business logic by enhancing the code reusability, and addressing the concerns of the end users regarding user interface and business logic are addressed at the corresponding layers. Three core components of this pattern are the model, view, and view model. The model receives updates from the view model and sends a notification. View model sends notifications to the view, which presents the output through data binding and commands. This arrangement also makes it possible to conduct isolated unit testing. The code-behind file serves the purpose of connecting the views to the view models. A key advantage of using MVVM is that user interface redesign is possible without affecting the code. The newer view becomes compliant with the current view model. The approach is an advanced methodology to encapsulate the business logic.

The beauty of the angular framework is that it allows adding tag attributes/directives into HTML pages. The attributes are used for two way data binding of the input and outputs. Installation of angular2 is a simple process when compared to other front-end application frameworks. It requires a minimum amount of skill at the in command prompt. First, install node.js and NPM from the official site https://nodejs.org/en/. One can choose among the different versions, so install a required one. NPM automatically comes with the node.js installation. If you are planning to start a new project, create a new project folder and then clone the angular 2 package from GitHub, present in the link here: https://github.com/buckyroberts/angular-2-template.git. It provides a proper template for the
angular project with the subfolders, index.html, and JSON files required for the project. Doing it this way, there are fewer chances of missing any important files in the project folder structure. Once this all is put into the new project, go to the project root folder path using the command line and run the following command:

```
npm install
```

This ‘install’ command will create a ‘node_modules’ folder. Download and install all the required project dependencies and libraries into it [22]. The src and src/app folder in the project has css, type script, javascript and .html files, which play a major role in project development. Once code changes are done in the files, the following command helps to run and test the project in localhost.

```
npm start
```

Once this command is run, the project instructions are executed in the localhost. The file should be saved, if further changes are made to the code. The server automatically refreshes to get the latest code changes. In any case, if the server needs to be stopped, type ‘CTRL+C’ on the keyboard and choose ‘Y’. Based on my experience in angular2, ‘npm install’ and ‘npm start’ are the two main commands that are to be noted.

4.2.2. **Web designing CSS –Bootstrap**

Bootstrap is an HTML and CSS based front-end framework used to develop website applications. It is an open-source framework and even supports JavaScript[23]. Originally developed by a small group of employees at Twitter, the impressive results enabled its widespread use. The interesting aspect about this framework is that Bootstrap comes in second
place according to the git-star ranking of the popularly used git-hub repositories [24].

4.2.3. IDE to develop - Visual studio code

Visual Studio, developed by Microsoft, is one of the most popular source code editors existing in the market for free. It has provided with numerous features and makes available multiple plugins. The features like GitHub, syntax highlighting with various colors, and node js support are particularly beneficial.

4.3. Details about the Website

The website is designed using the angular2 framework and Bootstrap styling features. It has four tabs: “home”, “instructions”, “finding sequences”, and “help”. Navigating through each tab redirects to its respective page, and the functionality of each page is discussed below.

4.3.1. Home Page

When the ‘npm start’ command is executed in the command line with no errors, the local host server is launched in the default web-browser setup by the user and opens the home page of the website. This page presents an overview of the entire site and describes the work done on each page. In the screenshot shown, clicking on each tab gives information about the page; for example, on selecting the instructions option, it describes the architecture details of 32-bit and 64-bit registers in the x86 processor.
4.3.2. Instructions

On the website, there is a 32-bit instruction tab and a 64-bit instruction tab. These tabs are provided to represent the bit-size and distinguish the different registers.

Design of the instruction page:

As shown in figure 8, the instruction page is mainly divided into five sections: 1) The Instructions list, 2) Input from the user, 3) Registers stack, 4) Memory stack and 5) About the instruction.

4.3.3. The instruction list block

In this thesis, the working of the assembly instructions of 32-bit and 64-bit is explained. Based on their functionality, the instructions are divided into five categories namely: Arithmetic, Bit-wise, Branching, Data movement, and Stack management/Functions.
Arithmetic instructions include Addition (ADD), Subtraction (SUB), Multiplication (IMUL), Division (IDIV), Increment (INC), Decrement (DEC). Bit-wise has Logical AND (AND), Logical OR (OR), XOR, NOT in it.

Branching section has more instructions when compared to others; there are unconditional Jump – (JMP) and Conditional Jump as follows:

Jump if Zero (or) Jump if Equal (JZ or JE),
Jump if Not Zero (or) Jump if Not Equal (JNZ or JNE),
Jump if Greater (or) Jump if Not Less than or Equal to zero (JG or JNLE),
Jump if Greater (or) Jump if Not Less than or Equal to zero (JG or JNLE),
Jump if Less than zero (or) Jump if Not Greater than or Equal to zero (JL or JNGE),
Jump if Less than or Equal to zero (or) Jump if Not Greater than zero (JLE or JNG).

A compare instruction (CMP) is also provided because all the above jump instructions need to be compared in the beginning before the actual jump takes place. Data movement instructions include Move (MOV) and Load Effective Address (LEA). The popular instructions PUSH and POP are included in Stack management. CALL and RET (Return) instructions are under the functions section.

4.3.4. Input block

In this area, the user will be able to give inputs for different types of instructions. Depending on the syntax and functionality of the instruction, the type and number of input boxes vary.

The instructions ADD, SUB, AND, OR, XOR, MOV, and LEA have five addressing modes in common – 1) Register-Immediate 2) Memory-Immediate 3) Register-Register 4)
Register-Memory and 5) Memory-Register. In the input block, these addressing modes are tested in two cases. One case deals with Register-Immediate and Memory-Immediate, and the other case allows the user to give the Register-Memory, Memory-Register, and Register-Register. So, in case one, there are two input boxes, one is a drop down, where the user can select a register or a memory address, the other box is a manual input. In case two, both the input types are provided as dropdowns for selection.

In the MUL instruction, the input has two sections, similar to the ADD instruction, but it has only five addressing modes. In section one, 1) Register-Immediate mode is present. In section two, 2) Register-Register, 3) Register-memory modes are present, and in the third section 4) Register-Register-Immediate and 5) Register-Memory-Immediate modes are present. When it comes to the DIV instruction, there are only two addressing modes. One is register and the other is memory. The logical NOT, INC (Increment) and DEC (Decrement) instructions have two addressing modes, similar to the DIV instruction. One is register mode and the other is memory mode. These instructions have one section in the input block. The Compare CMP instruction has four addressing modes. In section one, Register-Immediate is present, and in section two Register-Register, Register-memory and Memory-Register modes are present.

All the conditional jump instructions, JZ/JE, JNZ/JNE, JG, JNLE, JGE/JNL, JL/JNGE and JLE/JNG, follow the same addressing mode pattern. A CMP operation is done before JMP in all of these instructions. So, the compare operation is performed first, and then a corresponding JMP will be performed. Another point to be remembered for the JMP instruction is it does not have any addressing modes like in other instructions. Instead, there is a label or address of the instruction it has to go. When it comes to the input block structure, there are two sections. The first section has three inputs – the first input box for register or memory selection, the second
input for providing direct manual input and the third box for selecting the label or the address for performing the jump. The second section has three inputs – the first and second box for register and memory selection and the third box to select the label or address for performing jump. The label block for a conditional jump instruction is seen in figure 9.

The PUSH instruction has three addressing modes and two sections. In section one, the input box allows the user to enter an immediate value. In section two, an input box is provided to select register or memory. The POP instruction has two addressing modes similar to DIV, NOT, INC and DEC. One is register mode, and the other is memory mode. These instructions have one section in the input block. The CALL instruction is similar to the unconditional JMP instruction. The RTN instruction is created in such a way that it has the functionality of CALL and RTN. It has addressing mode just like in conditional JMP and has the option to select a label or address; therefore, it has only one input block for selecting the label or address.

![Image](image_url)

**Figure 9: Instructions page with Label column**

For each instructions, a button with its respective name is provided below its input box to operate the instruction. For example, the addition instruction has an ADD button present at the
bottom. On clicking it, the addition instruction is performed. Figure 9 displays the working of an instruction, with the change in its register value.

Figure 10: Instructions page with an example

4.3.5. Register block

The x32 bit page has 8 registers: EAX, EBX, ECX, EDX, ESP, EBP, ESI and EDI, whereas the x64 has a total of 16 registers where 8 of them are similar (RAX, RBX, RCX, RDX, RSP, RBP, RIP, RDI) and the additional eight registers are R8, R9, R10, R11, R12, R13, R14, and R15. An easy way to remember and avoid confusion among the names of these registers is that the 32-bit registers start with ‘E’ and 64-bit starts with ‘R’.

When it comes to the design of this block, a table is created with register names in the first column and their respective values are shown in the second column. Brown arrows point to the corresponding value of each register to avoid any confusion. Whenever there is a change in the register value, a green arrow points at the beginning of the register.
4.3.6. Memory block

This block is designed to have some fixed address memory elements and is similar to the register block. The first column has the addresses and the second column has the values stored in those addresses. Like mentioned for the register block, brown color arrows point to the corresponding value in the memory to avoid any confusion. Whenever there is a change in the memory value, a green color arrow is points at the beginning of the memory.

4.3.7. About block

This section is to provide information about each instruction, detailing how each instruction functions, the syntax of the instruction, and examples of representing the instruction. When we change from one instruction to another, the details are also changed accordingly.

4.3.8. Finding patterns

This page plays a vital role in contributing to this thesis. The purpose of this page is to load the input text file, read the contents, slice the crucial instructions and identify the sequences from the reduced instruction set.

This page has three main sections/blocks and it is displayed in figure 12.
In this section, the text input files are loaded. To the top left corner and just above this input block, there is an option to select the file from the system. On clicking the ‘Choose File’ button a window pops up displaying the system contents. Choose a required ‘.txt’ file with properly formatted assembly instructions. Once the file is selected, the textual contents are loaded into the input area. The purpose of this block is to display the original contents of the input text file provided by the user.

The required text file is supposed to have two columns; one has the address, and the other has its instructions. This information (address and instructions) has to be taken from an ‘.exe’ file. Choosing an ‘.exe’ file from the select file does not convert it to a ‘.txt’ file directly, because the ‘.exe’ file is very complex, and that is the reason debuggers are developed to decode them. Since the focus of the thesis is not to work on decompiling the ‘.exe’ file, a method of extracting the instructions into a text file from the exe is explained below.
To get the correct format of the input text file, follow the steps as mentioned. Open the required exe file using the x32dbg. Run the file and go to the beginning of the decompiled instructions. Select all the instructions using shift + end or by dragging the mouse until the end. Do not use ‘CTRL+A’ command, as doing this will modify the code and would not give the expected output. Paste the selected instructions into a .txt file, then copy the instructions from the text file and paste it in the ‘.xls’ template provided. Delete the columns other than address and instructions. The address will be in the first column, and the instructions will be in the third column. So, delete the remaining columns. Now copy all the contents from the ‘.xls’ to a ‘.txt’ file. The resulted ‘.txt’ file is the valid and formatted input file. This process is a little longer but has to be followed to get the required input file. Hundreds of test input text files with its ‘.exe’ and ‘.c’ files are provided along with this project on GitHub. They can be downloaded from GitHub into the system and the ‘.txt’ file can be uploaded directly into the input area.

Figure 12: Input text area in finding patterns page
4.3.10. Output instruction area

This output block/area is another text area similar to the input block. Once a valid input file is loaded successfully, click on the ‘Run’ button. As soon as this button is clicked, an algorithm is executed on the back end to limit the total number of instructions, and the final sliced/reduced instructions are displayed in the output area.

![Output area and results in finding patterns page](image)

Figure 13: Output area and results in finding patterns page

4.3.11. Results area

In this area, the final output data for different parameters are displayed. The output parameters are: total number of instructions in the input area, total number of instructions after limiting/slicing, total number of sequences identified, count of instructions under each sequence ID and displaying the identified sequences.
Figure 14: Results area

Figure 15: Tabular format of sequences in results area
4.3.12. Help

This section will provide the references and links that are helpful for learning assembly instructions. Clicking on a link will redirect the user to the relevant page.

![Figure 16: Help page](image)

4.4. Summary

This thesis developed a web-based interface to highlight different aspects of assembly language instructions. The author developed the code in Visual Studio IDE using the AngularJS programming language and Bootstrap for defining cascading style sheets (CSS). The website has four tabs, namely “home”, “instructions”, “finding patterns”, and “help”. The homepage provides an overview of the website and the features available on the other pages. On the instruction page, the user provides inputs for different instructions. The registers stack, memory stack, and the instruction details are displayed to the user. Finding patterns optimizes the instructions set by slicing the crucial instructions and identifying the sequences from the reduced instruction set. The help page provides links to learning assembly instructions.
5. Results, Conclusion and Future work

5.1. Results

In order to verify and validate the above-mentioned model, two steps are taken to ascertain that the assumptions made are rational and implemented properly with respect to software reverse engineering. Moreover, a model is also validated through assumptions, input parameter values and distributions and output values and conclusions [31]. However, in reality, it may be difficult to accomplish such a full validation of the model, especially if the system being modeled does not yet exist. This is the case with the WATSRE model. To my knowledge and the searches that were done in the internet, currently, there are no existing systems that can interactively show the working of assembly instructions, limit the debugged instruction count and list the meaningful assembly sequences.

According to Dr. Hillston’s lecture notes on ‘Model validation and verification’, he mentioned that the model verification is carried out through several ways which include antibugging, structured-walk through/one-step analysis, simplified models, deterministic models, tracing, animation and visualization, seed independence, continuity testing, degeneracy testing, and consistency testing [31]. For validation, he suggested that three approaches are used to validate the model: which commonly includes expert intuition, real system measurements, and theoretical results/analysis[31]. However, often any combination of them may be applied as appropriate to the different aspects of a particular model [31]. Each of these applicable methods is discussed below:

In One-step analysis (or) Structured-walk through, discussing the model with a friend or
an acquaintance, who is not aware of it before writing the document about the model, will help in identifying the issues present in it [31]. This method genuinely helped in rectifying the issues, and also paved the way for new ideas.

Representing the model in a graphical form in order to showcase its internal behavior accounts for ‘animation and visualization’ testing model [31]. Using this technique, dynamic behavior of the model is seen on the screen. Thus, if there are any bugs, they can be quickly identified. The WATSRE model has some animation icons and a nice graphic display, and any unexpected changes to these helped in identifying the errors easily.

‘Continuity testing’ includes testing the simulation models several times with various inputs (slightly modified)[31]. This test has been rigorously done on the WATSRE model using hundreds of input files, and it passed.

‘Degeneracy testing’ is a technique used to perform testing on an extremely large size of input data that is intended for the model [31]. And yes, large size data files were given as the inputs to the WATSRE model and it was found that more time is taken to load the data in the input screen. But, no drastic changes to the output were observed and it remained consistent.

‘Consistency testing’ is used to check if the model has the same features, even though the inputs keep changing and the outputs should not have any significant change [31]. Hundreds of files were tested in the WATSRE model to verify its consistency. This type of testing is performed for limiting the instructions and sequence comparison with multiple files.

Table 4 displays the model verification and validation techniques implemented on the WATSRE system.
When it comes to validation techniques, this thesis focuses on real system measurements and theoretical results/analysis. Real system measurements for instruction limiting are compared using the Godbolt compiler and the retargetable decompiler (RetDec). In addition to that, the percentage similarity of the WATSRE model was checked against RetDec. The Copyscape tool is used for finding the percentage similarity between the two files.

In this thesis, the instruction count is calculated for two types of files, one is programs without ‘functions’ and the other with functions. The final instruction count for the files that do not have ‘functions’ included has an average standard deviation value of +/- 3. On the other hand when it comes to the files with ‘functions' concept, the average standard deviation is +/-49.

<table>
<thead>
<tr>
<th>Model Verification</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-step analysis/ Structured walk-through</td>
<td>✓</td>
</tr>
<tr>
<td>Animation and Visualization</td>
<td>✓</td>
</tr>
<tr>
<td>Continuity testing</td>
<td>✓</td>
</tr>
<tr>
<td>Degeneracy testing</td>
<td>✓</td>
</tr>
<tr>
<td>Consistency testing</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model Validation</th>
<th>Validated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real system measurements</td>
<td>✓</td>
</tr>
<tr>
<td>Theoretical results (or) analysis</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4: Check for model validation and verification
Figure 17: Final instruction count comparison among different compilers for Type -1 files

For instruction limiting, a group of test files data for type-1 is compared and represented in the graph (Figure 18). Standard deviation is calculated for this data and the average value came out to be 2.37. These values show significant results and gives support to the algorithm for limiting the total instructions.

<table>
<thead>
<tr>
<th>Test_File</th>
<th>GodBolt</th>
<th>RetDec</th>
<th>WATSRE</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>72</td>
<td>66</td>
<td>67</td>
<td>3.21</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>23</td>
<td>23</td>
<td>4.04</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>49</td>
<td>49</td>
<td>1.73</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>0.58</td>
</tr>
<tr>
<td>6</td>
<td>58</td>
<td>55</td>
<td>53</td>
<td>2.52</td>
</tr>
<tr>
<td>7</td>
<td>63</td>
<td>67</td>
<td>60</td>
<td>3.51</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td></td>
<td></td>
<td>2.37</td>
</tr>
</tbody>
</table>

Table 5: Standard deviation calculation for Type -1
Figure 18: Final instruction count comparison among different compilers for Type -2 files

Similar to type -1, the instruction limiting is also compared to type 2 files and the graphs are plotted as shown in figure 19. The average standard deviation turned out to be a higher value than the type 1, and the value is 48.33. This is because of the ‘functions’ concept which is present in this type of files and due to this the assembly count in the compilers vary.

Significantly, the difference is not very high.

<table>
<thead>
<tr>
<th>Test_File</th>
<th>Godbolt</th>
<th>RetDec</th>
<th>WATSRE</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>315</td>
<td>274</td>
<td>320</td>
<td>25.24</td>
</tr>
<tr>
<td>2</td>
<td>518</td>
<td>359</td>
<td>618</td>
<td>130.62</td>
</tr>
<tr>
<td>3</td>
<td>114</td>
<td>89</td>
<td>127</td>
<td>19.31</td>
</tr>
<tr>
<td>4</td>
<td>458</td>
<td>345</td>
<td>440</td>
<td>60.72</td>
</tr>
<tr>
<td>5</td>
<td>119</td>
<td>89</td>
<td>106</td>
<td>15.04</td>
</tr>
<tr>
<td>6</td>
<td>473</td>
<td>403</td>
<td>408</td>
<td>39.05</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td></td>
<td></td>
<td>48.33</td>
</tr>
</tbody>
</table>

Table 6: Standard deviation calculation for Type -2
The difference of instructions count before and after applying the limiting algorithm is shown in the figure 20. As observed, the difference is so huge such that the final count is too minute to see in the graph. This reason accounts for the 3d view of the graph shown. Table 6 with test data files with numbers is also included in the graph.

![Figure 19: Instruction count before and after limiting algorithm](image)

When it comes to evaluating the sequence identification, there are no exact comparison tools available. The closest comparison can be done with the Godbolt decompiler and the theoretical knowledge to identify sequences is gained from an x86 disassembly book. Moreover, theoretical analysis using an extensive number of files is also done for the same.

For all the sequences identified, matching percentage is checked with Godbolt. The matching values are divided into four types: No match (0%), Partial match (50%), Satisfactory match (80%), Complete match (100%). The percentage of these four categories is depicted in a pie-chart (figure 21). It can be observed that more than 50% of the match is satisfactory, 33% is a complete match, and only 11% comes under partial match. The satisfactory match is primarily
due to the minimal change in the instructions names in Godbolt and the reason for this is Godbolt is a 64-bit instruction compiler and WATSRE finds the sequences in 32-bit instructions. Though the meanings are same, there are slight differences in the names of the instructions.

Figure 20: Comparison of Sequence matching with Godbolt

After the sequences are identified, the count of each sequence ID is calculated. The figure 21 represents the count of sequences for a test file. This helps in observing the more and less frequent sequences.
Figure 21: Histogram with sequence ID in x-axis and Count on y-axis

5.2. Conclusion and Future work

This thesis presented a model named ‘WATSRE’ for alleviating the learning process of assembly language for software reverse engineers. This model or system not only teaches the instructions, but also helps in memorizing the assembly instruction sequences. Alongside, this system will limit thousands of assembly instructions to the required number present in the C file. By using this system, a novice in reverse engineering will be able to autodidact oneself the assembly instructions and learn how a group of assembly sequences translate into meaningful words/ high-level language. The WATSRE model is verified in terms of its percentage similarity against the Retdec and Godbolt decompilers. The prime reason to compare WATSRE with Retdec instead of Godbolt is because it acquires addresses, institutions, and opcode. Apart from all these, this research helps in perceiving the importance of assembly language in reverse engineering.

Reverse engineering fused with assembly creates a vast and complex platform with tremendous opportunities. The current model is an initial system to detect the sequences, and is
modelled for finding 45 sequences. Currently, 50 sequences in 7 concepts are identified in this system from which 45 sequences are identified using the ‘WATSRE’ model and the remaining are theoretical because of their unpredictable nature. The count can be elevated in the future by incorporating more concepts which in turn elevates the number of sequences. Furthermore, this research can also be extended across various compilers and processors. When it comes to the user interface and visualization, there is a lot of future scope in this area. Enhancements include highlighting the identified sequences in the output window similar to the Godbolt compiler, adding more instructions to train the assembly instructions, feeding an executable file as a direct input instead of as a text file, definitely styling the icons and resizing the website pages and more.
**BIBLIOGRAPHY**


[10] Adam Smith, Robert Mills, Adam Bryant, Michael Grimaila and Gilbert Peterson (200X); the Role of Expert Systems in Reverse Code Engineering Tasks.


