Final Year Project Report

Full Unit – Final Report

A Disassembler for Reverse Engineering Binary Executables

Gurjan Lally

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Professor Gregory Chockler

Department of Computer Science

Royal Holloway, University of London

Table of Contents

Abstract

Disassemblers are powerful tools, used for many purposes. One such purpose is the Reverse Engineering and analysis of Binary Executables. That is, scrutinizing the internal functionality of some program compiled as an Executable file containing some binary or machine code. To actually perform the analysis, Reverse Engineers cannot feasibly analyse Binary Executables with just the executable and no other tools. A higher-level abstraction from its Machine Code format is required. Such higher-level abstractions include Source Code and Assembly Code. Disassemblers provide the means to decode machine code, presenting interpretable assembly code. Disassemblers perform the inverse operation to that of an assembler, most often with output in a human-readable format and unsuitable for direct input to an assembler. This is what primarily makes a disassembler a Reverse Engineering Tool.[1] In addition to this, it makes verifying the output of a disassembler most difficult.

This project aims to produce a state-of-the-art disassembler, with core functionality being the graphical representation of output disassembly code from some input file. Additional important functionality of disassemblers in general involes means to represent the control flow graph of the given executable graphically, data export, resillience against binary obfuscation methods, and supporting various file types and architectures.

The Disassembler in this project makes use of the acclaimed Capstone Engine, a powerful ‘engine’/library, which is a level of abstraction below the Disassembler itself. The Capstone engine is used to actually decode instructions from bytes passed into it and feedback disassembly code. Capstone itself cannot take an input file and return accurate, meaningful

‘disassembly code’. Capstone will do this only for bytes passed which correspond to executable code/program instructions.

Motivation

Upon reading the original project specification, I was shrouded with fear of the associated concepts, including everything surrounding executable files (format, how they’re executed, compilation process), Assembly code, and Reverse Engineering. I had no idea what to expect from the project, apart from the fact that it would be a challenge, as well as a perfect learning opportunity.I did not even know what a Disassembler was.

Undertaking this project will be incredibly useful going forward after university, due to the invaluable concepts studied and the complexity of them. To independently combat such complexity has given me insight into how to handle similar situations in the future. Furthermore, the work carried out has shown me how to develop a piece of software effectively in terms of managing a repository, outlining requirements, and devising program structure.

Chapter 1

Introduction

This chapter outlines the project rationale, covering the project specification, aims, objectives, and necessity for the software product produced.

1.1 Project Specification

‘To analyse and reverse engineer compiled binaries, it is necessary to disassemble the instructions that will be executed at runtime. A disassembler reads a target binary (ELF for Unix systems or PE for Windows) and determines the entry point from the file header. It then begins decoding instructions beginning at the entry point, in either a linear sweep (i.e., a straight sequence) or a recursive traversal (i.e., following jump targets). Determining jump targets can be hard, so practical disassemblers often also use various heuristics to discover code. The decoded instructions are arranged in a graph that can be drawn or exported in different formats for further analysis.

This project will build a disassembler in Java using the Capstone library, a framework for decoding instructions from multiple architectures. The Java disassembler will have to implement the file parsing, traversal, heuristics, data export and visualisation.’

1.2 Project Aim

The aim of the project is to build a state-of-the-art disassembler using the Capstone library. Associated with this main aim is to implement the Recursive Traversal algorithm, which is required to obtain overall useful and meaningful disassembly results.

1.3 Necessity
Disassemblers are required for the analysis of software which has been compiled to executable machine instructions, where source code is typically not available. At times, this can be frowned upon, as programs are not meant to be Reverse Engineered once compiled. The positive use cases, however, are instances such as malware analysis, where the analysis of malicious software is performed to understand the behaviour in order to mitigate its effect.

Reverse Engineering also has many other use cases, such as Vulnerability Research & Development, Digital Forensics, the Detection of Plagiarism in Software, and Debugging. Disassemblers are integral tools to carry out these activities and, as such, there is an industry requirement for top quality disassemblers.

1.4 Objectives

- Understand the process of Disassembly and its link to Reverse Engineering
- Gain an understanding of theoretical concepts and challenges associated with the disassembly process
- Build the Disassembler, giving a visual representation of disassembly to the standard of already available disassemblers
- Evaluate the results of disassembly (verifiability)

1.5 Report Structure
This report consists of 10 chapters.

- Chapter 2 covers the Legal aspects and Professional Review of Reverse Engineering

- Chapter 3 covers the theory associated with disassembly, introducing it from its most basic description, to complex challenges associated

- Chapter 4 gives an overview of the Capstone disassembly engine, introducing it, explaining its widespread adoption, and usage.

- Chapter 5 outlines the basics of parsing ELF files

- Chapters 6-9 explains the software produced in depth including functionality, testing, implementation decisions, and software engineering practices

- Chapter 10 consists of appendices being the project diary and the User Manual

Chapter 2

Legal and Professional Issues in Reverse Engineering

The Reverse Engineering of software has many applications. This section discusses the legal aspect of Reverse Engineering and professional issues associated with activities which Reverse Engineering and Disassembly is integral to.

2.1 Introduction to Reverse Engineering

The process of Reverse Engineering software arises from the fact that we typically do not have access to a piece of software's source code, and only its corresponding binary if we want to understand its internal functionality. The Reverse Engineering of Software involves taking a piece of software and heavily scrutinizing its internal functionaliy for one of many possible purposes. Such purposes could be malicious by intent or completely innocent, perhaps driven by curiosity. One may want to reverse engineer some software in order to derive architecture and design information, to the degree that it can be modified or reproduced in another independent software work. [11]

There are a range of professional activities which the Reverse Engineering of software can be integral to at times. Such activities include Malware research, software plagiarism detection, Digital Forensics, Assisting the modification of legacy programs, Vulnerability Research & Development, and Binary Patch Development for unmaintained software.[12] All of these uses for reverse engineering are positive use cases, with typically no ethical issues surrounding them should the parties carrying out the activities have the legal permission to do so, and the purpose remaining innocent.

From a legal perspective, we are concerned with the US Copyright act of 1976 and the Digital Millennium Copyright Act, as well as the End-User Licence agreement; all of which are outlined below. From an ethical perspective, we are concerned about the impacts of Reverse Engineering on people, as well as the responsibilities which professionals must be conscious of that come with Revere Engineering.

2.2 Legal Issues

The US Fair Use Copyright act of 1976 states[15]:

The fair use of a copyrighted work, including such use by reproduction in copies or by any other means specified by that section, for purposes such as criticism, comment, news reporting, teaching (including multiple copies for classroom use), scholarship, or research, is not an infringement of copyright. In determining whether the use made of a work in any particular case is a fair use the factors to be considered shall include—

1. the purpose and character of the use, including whether such use is of a commercial nature or is for nonprofit educational purposes;
2. the nature of the copyrighted work;
3. the amount and substantiality of the portion used in relation to the copyrighted work as a whole; and
4. the effect of the use upon the potential market for or value of the copyrighted work.

The fact that a work is unpublished shall not itself bar a finding of fair use if such finding is made upon consideration of all the above factors.

By interpreting the fair use act in the context of Reverse Engineering, we can assume that it largely supports Reverse Engineering performed for academic purposes, as opposed to commercial purposes. In fact, the act is largely concerned with commercial impact, factoring in 'effect of the use upon the potential market for or value of the copyrighted work' when determining fair use. It is also clear that fair use acts as a potential point to reference for Reverse Engineers to defend their activities if legitimate.

The US Digital Millennium Copyright Act was also introduce in 1998 and introduced the following laws related specifically to Reverse Engienering Software[13]:

1. Notwithstanding the provisions of subsection (a)(1)(A), a person who has lawfully obtained the right to use a copy of a computer program may circumvent a technological measure that effectively controls access to a particular portion of that program for the sole purpose of identifying and analysing those elements of the program that are necessary to achieve interoperability of an independently created computer program with other programs, and that have not previously been readily available to the person engaging in the circumvention, to the extent any such acts of identification and analysis do not constitute infringement under this title.
2. Notwithstanding the provisions of subsections (a)(2) and (b), a person may develop and employ technological means to circumvent a technological measure, or to circumvent protection afforded by a technological measure, in order to enable the identification and analysis under paragraph (1), or for the purpose of enabling interoperability of an independently created computer program with other programs, if such means are necessary to achieve such interoperability, to the extent that doing so does not constitute infringement under this title.
3. The information acquired through the acts permitted under paragraph (1), and the means permitted under paragraph (2), may be made available to others if the person referred to in paragraph (1) or (2), as the case may be, provides such information or

means solely for the purpose of enabling interoperability of an independently created computer program with other programs, and to the extent that doing so does not constitute infringement under this title or violate applicable law other than this section.
4. For purposes of this subsection, the term ‘interoperability’ means the ability of computer programs to exchange information, and of such programs mutually to use the information which has been exchanged.

We can gather from the above the terms under which a piece of software may be Reverse Engineered. The key factor is that Reverse Engineering is considered acceptable if the purpose is to achieve ‘interoperability’ between an independently created program with another program. Additionally, it states that reversing a given program should be a necessity to achieve this. It should also be noted that Fair Use exists in line with the above, meaning that careful considerations must be made based on the situation.

End-User License Agreement

An End-User License Agreement is a legal contract between a software manufacturer and user, outlining the terms under which users may use the software, giving a list of conditions of what the user may and may not do. It can state anything from the number of copies that can be made to conditions under which it can be reverse engineered.[14]

A key point to be made about the End User License agreement is that it can explicitly override conditions laid out by the previous two Copyright Act additions, due to the fact that it is a legally binding **contract**. This means that if an EULA states that a piece of software must not be reversed, it absolutely must not be reversed, as Fair Use is ignored if this is specified in the EULA. The contract’s terms hold priority over other legislations.

We can therefore gather that if some professional wishes to Reverse an Application, no assumptions should be made, and an expert in law should be consulted on the matter to avoid any risks. The laws are extremely situationally dependent so it makes good sense to do this – especially if any market impact on a software product can be associated with the Reverse Engineering of some other product.

2.3 Professional Issues

There are some professional issues associated with disassembly and Reverse engineering. Given the legal aspects of Reverse Engineering, we consider potential scenarios which are the reason such laws exist in the first place. It is futile to assume that use cases for Disassemblers do not include Reverse Engineering with malicious intent, since they are highly supplementary to this.

Reverse Engineering with malicious intent is the use of associated tools (including Disassemblers) and methodologies, coupled with expert knowledge in order to understand the operation of a piece of software and thereafter abusing this understanding to do something unintended from the software’s original specification. The aforementioned understanding of a piece of software will go beyond the original level understanding expected from the software manufacture.

Abuse of such understanding could be used to find hidden features of a software product, modifying an existing product, 'cracking' software products, or developing malware based on information learned.

As an example, an application's license key validation algorithm could be reverse-engineered, allowing the construction of what are widely known as 'Keygens'. Keygens provide means to generate a product licensing key, such that an illegitimately obtained piece of software can be activated for use in a legitimate way. This is known as 'cracking', mentioned previously. Not only is this illegal in most cases, but it is highly unethical. Since most paid pieces of software require a validation key, cracking paid software offers a way around the validation key, so that paid software can be used for free. There will always exist users who would much rather obtain a free piece of software in this manner than pay the rather expensive current prices of software, should the opportunity arise. This directly impacts software manufacturers' revenue, since there might be less paying users on the market. This highlights the unethical aspects of Reverse engineering, since a software manufacturer may run bankrupt as a result of this form of software piracy.

There may also be cases in which Reverse Engineering with innocent intent poses an indirect threat to regular individuals. For example, a perfectly legitimate Reverse Engineer may be performing software vulnerability research and exploit development for presentation to the relevant software manufacture, and thereafter, publication. Unauthorised users could compromise the individual's system and discover the vulnerability in this time. An unauthorised individual may leak the vulnerability or develop a malicious piece of software which exploits the vulnerability, and release it onto the wider internet. In this case, Reverse Engineering is seen to still poses a threat even when unintended to. A similar situation occurred when vulnerabilities harvested and stored by the NSA were leaked.[16] It is therefore a Reverse Engineering professional's duty to keep information learned through the process undisclosed. Information obtained should only be disclosed once discussed and agreed with the original software manufacturer.

Reverse Engineering professionals should not participate in the previously mentioned malicious or unethical activities. They should also have full permission to Reverse Engineer pieces of software, or may face legal consequences. Furthermore, it is their full responsibility to still obey the IT professional code of ethics and conduct, ensuring that potentially impactful information discovered from Reverse Engineering a piece of software should be reported straight to the original software manufacturer. Professionals in this field may sometimes be requested by individuals of higher authority in an organisation to perform tasks that are inherently unethical, and they should not participate in doing so. Such a situation has been seen in [17], whereby Atari reverse-engineered a Nintendo product in order to create a product which defeated that Nintendo product. While the legal outcome found Atari to have not committed any wrong-doing, reversing the Nintendo product was a highly unethical task performed by the group of Reverse Engineers, which should not have performed the analysis.

It is important in this project to keep in mind the role of Disassembly in the Reverse Engineering process, and how it can contribute to some of the dark activities mentioned above. Undoubtedly, individuals involved in these activities are using disassemblers, which means that the software product being produced in this project may end up being used for unethical or illegal purposes if adopted. It is obviously unethical on my part to let malicious users potentially cause harm to others through information about a piece of software obtained through the Disassembler produced.

Chapter 3

Disassembly

This chapter outlines key theoretical concepts related to the the process of disassembly, learned over time in order to understand how to actually produce a disassembler.

3.1 Introduction to Disassembly

Disassembly is apart of the process of Reverse Engineering applications. At its simplest, Disassembly is the process of recovering a sequence of Assembly code from 'Machine Code' instructions. The next logical step in the Reverse Egineering process after disassembly would be decompilation; the inverse of compilation (source code to machine code), which aims to recover higher-level abstractions from the assembly code. The project itself includes an element of decompilation being the 'Control Flow Graph' (section **3.7**), but ultimately involves implementing the means to disassemble.

Machine code is interpretable by a computer's microprocessing unit, and consists of sets of instructions completely dependent on the CPU architecture. For example, Intel has its own instruction set, which has been updated consistently over the years as their processors have developed. Conversely, ARM has its own entirely different instruction set. Assembly language is arguably the lowest level huamn-interpretable programming language, and is one of the lowest level abstractions of machine code. This makes it incredibly valuable to know for Reverse Engineers, as it gives a direct representation of a program's execution, including the tracing of registers and values passed to function calls.

As per Intel's instruction set, an instruction can be anywhere between **one** and **fourteen** bytes. A typical Intel instruction, for example, will contain an instruction Prefix, used to provide section overides, perform bus lock operations, and to change operand and address sizes.[2] An intel instruction will also contain Opcode, which is a 1-2 byte instruction. The highest level view of Opcodes is that they most commonly tend to either manipulate values in registers, manipulate control flow, or compare values in registers. Their functionality is abundant, however, going far beyond this. Finally, typical instructions will contain (but are by no means limited to containing) an Operand. Operands are the actual values affected by Opcodes. Most often, these are the registers/addresses.

It is a Disassembler's job to parse executable files, and to decode every byte in streams of executable Machine Code within the file. They do this by taking into account every byte's value, referencing rules set out by the instruction set's designers to define what executable instruction bytes look like, and their meaning.

Despite the fact that an executable contains source code compiled to a series of Machine Code instructions, the contents of an executable are not solely executable Machine Code, and they contain many other sections. The *data* section of an executable is an example of this, containing a program's static variables, of a fixed size. The *text* section of an executable file, for example, will contain executable code, which is what a disassembler is concerned in dealing with to give meaningful output representing a program's execution. This means that the executable must be parsed to determine the beginning of the *text* section if the executable were to be disassembled. A more common term for the beginning of the *text* section and transitively the location of executable machine code is the **entry point**.

3.2 Correct Disassembly

The art of disassembly is not an exact science. Disassembly tends to give mixed results with regard to approximating a program's execution. Often, disassembly can provide an over-approximation of a program's execution, disassembling everything in the text section (and perhaps more by mistake). This can make it tricky to perform binary analysis, as analysts are led to believe in some execution flow of a program which may deviate from the actual flow.

Conversely, disassembly can provide an under-approximation of a program's execution. This is especially common in instances of obfuscated disassembly (see 2.6), where the purpose is to hinder the discovery of code by disassemblers. This can lead to entire regions of executable machine code instructions not being disassembled, resulting in them being omitted from control-flow representation.

This is obviously a serious problem when a deep understanding of a particular executable is required, as an accurate representation of the binary is not provided. This means that malware analysis or threat identification may transitively produce inaccurate or incorrect results.

3.3 Static and Dynamic Analysis

Once the entry point of an executable file has been determined, it can be disassembled. There are two main approaches to disassembly: **Static** disassembly and **Dynamic** analysis. Static disassembly involves examining the executable file and producing some disassembly code, but not actually executing the file during the process. Static disassembly deals with the file that it has been given, and performs analysis accordingly, unless the disassembly might have access to external libraries outside the scope of the executable. Its advantage is being able to process the entire file all at once. Dynamic analysis involves disassembling a slice of a program. This could be based on some particular input, where the execution is monitored by an external tool, such as a debugger, to identify instructions executed. Dynamic analysis therefore also means executing the program in question.

Static disassembly takes time proportional to the size of the program to execute, given that it analyses an entire machine code file. Dynamic analysis takes time proportonal to the number of instructions executed during the runtime disassembly process, and takes time many factors greater than Static disassembly.[3] This project is only concerned with static disassembly and considers some approaches to static disassembly.

3.4 Static Disassembly Algorithms

There are two main algorithms associated with static disassembly, being the **Linear Sweep**, and **Recursive Traversal** algorithms. In addition to these two algorithms, many other algorithms exist as extentions, combinations, or both an extension and combination of the two algorithms.

Linear Sweep

The Linear sweep algorithm begins at a program's entry point and disassembles the entire text section. It disassembles each instruction as and when encounted, until an illegal instruction is encountered. The main problem with this is that it is mistake prone. The linear sweep assumes that the entire section being disassembled is executable machine code, and does not take into account, for example, any data embedded in the instruction stream. Any data encountered is simply disassembled as if it were just another byte in a stream of executable machine code. For example, some embedded data could exist (such as null bytes for padding) in direct succession to a jump instruction pointing further ahead, out of the embedded data's address range. When executed, the data section will be skipped as a result of this jump instruction. Had the disassembly been accurate, the data would not have been disassembled in the first place, and the disassembly would continue from the jump location. However, this is not the result of the linear sweep algorithm. The linear sweep algorithm will instead disassemble the jump instruction, the embeeded data in succession to it, and anything after that. Worse still, this will go unnoticed most of the time, apart from certain special circumstances (such as when an invalid opCode is detected)[4]. With the difficulty of verifying disassembly code produced as output, this can be very frustrating.

Recursive Traversal

The static Recursive Traversal algorithm is an effective algorithm, which eliminates the linear sweep's shortcomings. This algorithm takes into account the destination address of branch instructions (such as Jump or Call), and proceeds to disassemble at that address whenever one is encountered. If, when disassembling at the target address, another branch intruction is encountered, the disassembly function is called at the destination address and so on. So, the disassembly is recursive, and is termed as such, but only different from a linear sweep in this way. Its key feature is the elimination of the disassembly of data embedded in executable code areas. As a general note for recursive traversal, the next processed byte in the executable file depends on the result of the last processed instruction.[5]

For Call Instructions and Conditional Transfer Instructions, i.e. branch instructions that only jump to some location if a condition is met, these instruction will have two disassembly targets, being the destination address and the next proceeding instruction from the Call/CTI. This differs to unconditional jump instructions, which only have the disassembly target specified by its operand.

3.5 Obfuscation to Thwart Recursive Traversal

Both the linear sweep and recursive traversal algorithms can be tricked by a variety of obfuscation methods, which could be purposefully implemented by hand-crafted assembly code before compilation or potentially binary instrumentation by code injection thereafter. Obfuscation methods could render entire disassembly algorithms useless if they rely on assumptions made by the algorithm. This is because obfuscation will often hide the invocation of functions and subroutines, meaning that entire regions of executable code may never be discovered.

The Recursive Traversal algorithm itself has some shortcoming, which can be exploited by obfuscation mechanisms. The most obvious shortcoming is that functions and subroutines which are not ever explicitly called during the program's execution may never actually be disassembled. While this may be representative of the program's true execution, it is not ideal for the vast majority of scenarios. The most common example of this arises from functions called by calculating their address during program execution. Recursive traversal algorithms most often fail to take this into account, especially if the calculation of the target address is related to some program input.

Recursive traversal algorithms may also have difficulty when dealing with indirect jump calls. For direct jump calls, the target address is encoded alongside the jump instruction. This makes it simple enough for the recursive algorithm to set sights on disassembling at the operand of the jump instruction. Indirect jump calls, however, use a target address held in some general purpose register or memory location. This is of course unknown at the time of disassembly for basic disassemblers, so it is a major hinderance. Despite this, various heuristics exist to approximate indrect jump target addresses, and disassembly at all possible addresses. Such heuristics often result in failure themselves: not so much in their inability to approximate addresses, but the fact that approximated addresses might just happen to land in the middle of some address – resulting in incorrect disassembly[4].

Branch functions also present another big problem to the recursive traversal algorithm. Jump instructions can be replaced by calls to functions which manipulate registers, such that the original target of the jump instruction can be specially positioned so that when the function returns, it returns to target of the jump instruction instead of to the original execution flow. Recursive traversal disassemblers will not know this, and simply assume that, when a return instruction for a function is reached, diassembly should continue at the instruction proceeding the original call to the branch function.

Given that recursive traversal algorithms assume that Call/CTI instructions have two targets (proceeding instruction and branch target), code can be crafted such that bytes proceeding these types of instructions are junk bytes. This results in the junk bytes being disassembled, poising the disassembler to misinterpret future instructions.

3.6 Combating Obfuscation

The results of disassembly after a method of obfuscation which results in proceeding bytes being incorrectly disassembled will only be incorrect up until a certain point on most architectures. This is due to the fact that the disassembly will naturally re-synchronize with the actual instruction stream if the results produced are only off by a few bytes. This means that methods of obfuscation must take this into account.

However, self-repairing disassembly is not at all a reason to ignore obfuscation, as the problems introduced by the obfuscation are still present, even if only affecting part of the instruction stream. The impact can still be massive, especially if obfuscation mechanisms are abundant in some binary.

Disassemblers such as IDA pro tend to not fail silently, offering a list of failures encountered during the diassembly process. When failures occur, they are presented to the user. However, it is up to the users to determine how to fix any failures, such that a newly disassembed version will be correct. An example of a solving a failure is patching bytes in an executable, perhaps replacing or modifying them, which is non-trivial. A nice approach offered by IDA is to mark certain bytes as text or data and get a newly updated version of the disassembly after this has been done. This can be particularly useful in instances such as when dealing with hand-crafted obfuscation methods designed to unconditionally jump in the middle of subroutines, or junk bytes proceeding a call instruction. But again, it is down to the user to determine junk bytes – something which requires experimentation and extensive analysis.

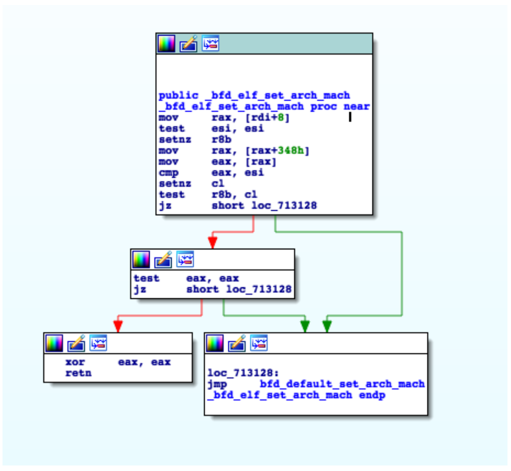
There are several proposed algorithms to deal with methods of obfuscation, but very few algorithms which are capable of dealing with varieties of obfuscation mechanisms, and for instances where little assumptions can be made about information available related to a binary. [4] for example, outlines an effective algorithm working as a hybrid of the linear sweep and recursive traversal algorithms. The algorithm is effective for identifying and dealing with jump tables appearing in the text segment of a an executable, but the algorithm is reliant on relocation information for executables, which is not always available as stated in the paper.

[10] presents an algorithm with perhaps the best results of any algorithm, disassembling with ~90% accuracy on obfuscated binaries with obfuscation tool specific knowledge. Additionally, the algorithm still performed better than the hybrid algorithm devised in [4] even without obfuscation tool-specific knowledge. This algorithm is actually optimal due to the fact that its non-tool-specific version can operate under the most basic of assumptions: valid instructions must not overlap, conditional jumps can either be taken or not taken, an arbitrary amount of junk bytes can be inserted at unreachable locations, and that control flow does not have to continue immediately after a call instruction.[10]

Such algorithms seem to be the best way to deal with obfuscation in executables. The recursive traversal and linear sweep basic algorithms will not persist against obfuscated executables how these algorithms do. As a side note, it would be an interesting task to perhaps attempt the implementation of [10] in the future.

3.7 Control Flow Graph representation

Control flow graphs are critical in the reverse engineering process in order to understand program semantics. They normally represent blocks of instructions ending with a branch instruction and none in-between, known as 'Basic Blocks'. They help to visually assist the identification of common instruction sequences to the trained eye and a provide general hierarchical view of program subroutines and function calls. They are therefore most often represented by a hierarchical graph layout, with each function of a disassembled program maintaining their own control flow graph. The root node of a hierarchical CFG typically represents the first instructions of a function, with the end node representing the final instructions. Figure 0 shows a basic example of this.



```
public _bfd_elf_set_arch_mach
_bfd_elf_set_arch_mach proc near
mov     rax, [rdi+8]
test    esi, esi
setnz   r8b
mov     rax, [rax+348h]
mov     eax, [rax]
cmp     eax, esi
setnz   cl
test    r8b, cl
jz      short loc_713128
```

```
test    eax, eax
jz      short loc_713128
```

```
xor     eax, eax
retn
```

```
loc_713128:
jmp     bfd_default_set_arch_mach
_bfd_elf_set_arch_mach endp
```

Figure 0 – IDA Pro control flow graph representation for a function

Control flow graphs can be built at different stages of the disassembly process. Some disassemblers are known to perform a linear sweep first, and then form a control flow graph after. The control graph is therefore formed from the disassembled instructions but prone to being incorrect due to the weaknesses of linear sweep. Some other disassemblers do the same but using the recursive traversal algorithm, and some construct the CFG *during* recursive traversal.

Through building the control flow graph during the recursive traversal process, we may not have a complete and correct representation of the control flow of an executable. This is due to the fact that the recursive traversal itself might produce an under-approximated representation of a program's executable instructions. On the other hand, the linear sweep could help to produce a more complete control flow graph, as the entire .text section would be disassembled, meaning that building a CFG afterwards would have more nodes overall than a CFG produced by recursive traversal. But again, relying on the linear sweep is much more likely to produce incorrect disassembly results than recursive traversal.

Building a correct control flow graph is therefore just as tricky as obtaining correct disassembly when doing so by the above methods, especially as the process of obtaining a

CFG can be vastly similar to how the Recursive Traversal algorithm operates. However, it should be noted that there are alternative ways to build a control flow graph, as mentioned in [10]. The method of doing so in [10] works on the basis of disassembling **every** byte in an executable, and extracting all control transfer instructions, such as jump instructions or call instructions. An initial control flow graph is then built from these CTIs, with the assumption that a true representation of the program's CFG is a sub-graph of the initial CFG. A set of rules is then imposed on the initial CFG to obtain the sub-graph.[10] Such a method far out-performs the classical method of building a control flow graph in ordinance with a disassembly algorithm/

Chapter 4

Overview of Capstone

This chapter covers the background, internal functionality, and capabilities of the Capstone Engine.

4.1 Introduction to Capstone

Tools assisting exploitation development, debuggers, and disassasemblers themselvs sit a layer of abstraction above what are known as 'Disassembly Engines'. Disassembly engines/libraries are responsible for decoding encoded instructions passed in, in order to return some correct assembly code. Disassembly Engines must account for every possible machine code instruction, which is absolutely dependent on the CPU architecture. It is therefore a challenge to have a Disassembly Engine which can work for multple CPU architectures, given the complexity of such a task. Decoding machine code instructions is, for a human at least, really a very complex task. Capstone, however, is a Disassembly Engine which can achieve this to a high standard.

4.2 Capstone Engine Internals

The Capstone project itself uses a modified fork of the 'LLVM' project, a set of frameworks to build a compiler. Included in the 'LLVM' project is a disassembler, known as the Machine Code Module. The background decision to choose the Machine Code Module was the fact that it is being maintained most by the developers of each CPU architecture. That is, Intel (x86 architecture), Arm and Apple (Arm + Arm64), Imgtect (MIPS), and others. This means that new instructions and bug-fixes are implemented incredibly frequently. The biggest shortcoming of using the Machine Code Module is that it is written in C, not designed for thread safety, nor designed for windows. Capstone works on the basis of taking the disassembler core and making minimal changes. [5]

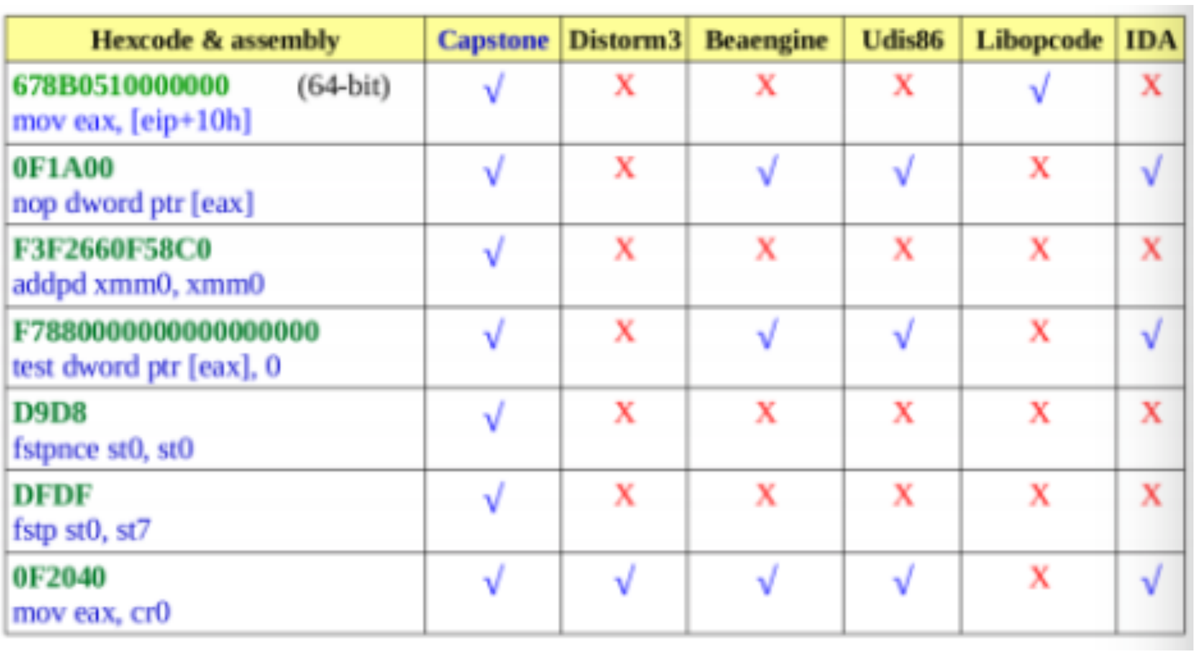
4.3 Features and Capabilities

Capstone satisfies the industry requirement of working on multiple architectures, sucessfully decoding instructions from x86, Arm, Arm64, MIPS, PPC, and 4 more architectures.

Additionally, Capstone also works on multiple platforms, including MacOSX, Linux, iOS, BSD, Android, and Solaris. Capstone, as a key feature, has the ability to also deal with much more tricky x86 instructions than alternative disassembly engines. An example of some of these instructions are shown in figure 1. At the time of release in 2013, no such other disassembler even came close to satisfying these requirements. [5]

Where Capstone prioritises covering tricky and corner cases of code, the Machine Code Module of LLVM does not, and as such functions as the lesser disassembly engine, all (other) things considered. At the time of development, Casptone could also handle all notable x86 malware tricks. [5]

Other powerful features of the Capstone Engine include the ability to retrieve important semantics of disassembled instructions, such as a list of implicit registers read and written. For disassembled instructions, it can also be noted if the instruction belongs to a group of instructions.[5



Hexcode & assembly	Capstone	Distorm3	Beaengine	Udis86	Libopcode	IDA
678B0510000000 (64-bit) mov eax, [eip+10h]	√	X	X	X	√	X
0F1A00 nop dword ptr [eax]	√	X	√	√	X	√
F3F2660F58C0 addpd xmm0, xmm0	√	X	X	X	X	X
F7880000000000000000 test dword ptr [eax], 0	√	X	√	√	X	√
D9D8 fstpnce st0, st0	√	X	X	X	X	X
DFDF fstp st0, st7	√	X	X	X	X	X
0F2040 mov eax, cr0	√	√	√	√	X	√

Figure 1 – tricky x86 instructions handled by Capstone

4.4 Project and Capstone

The Capstone Library is used in the disassembler, and is responsible for decoding sequences of bytes supposedly representing instruction bytes. It is used as a shared object in the project, invoked to disassemble instructions of the 64-bit Intel instruction set in its simplest (and fastest) mode of operation, ‘detail off’. This mode simply gives us instructions’ most basic details including size, address, and string representation.

4.5 Community and Products Utilising Capstone

As an open source project, Capstone receives plenty of contribution overall – with over 1,100 issues either closed or open on its GitHub page. The owner is fully commited to Capstone, declaring ‘shame’ on the industry for not providing a solution similar to Capstone before its 2013 release. The project’s lifecycle is guaranteed by the owner, further highlighting commitment to the project.

In addition to this, according to Capstone’s showcase webpage, over **371** products use the Capstone engine internally. It seems a trustworthy and widely adopted project, most suitable for use in this project.

Chapter 5

Parsing Binary Executables (ELF)

This chapter explains the process involved in parsing an ELF file, resolving information suitable to the task of disassembling its executable instructions. Extracting information such as the ELF’s symbol table is the first step towards producing a more human-friendly visual representation of disassembly.

5.0 Introduction to ELF Files

ELF files are one of many possible executable file formats, produced perhaps as output from the compilation process of source code on Unix systems, or manually. As previously mentioned, to produce accurate disassembly of some executable, the entry point must be determined. That is, the beginning of the section of machine code in an executable that can actually be executed by a CPU of the relevant architecture. This is due to the fact that executable files do not solely consist of information useful to a disassembler to produce meaningful output.

5.1 Determining ELF Entry Point

To actually determine the entry point, the ELF files must be parsed. Due to the layout of ELF files, they provide means to do this swiftly programatically. The first four bytes of any ELF are 0x7F, 0x45, 0x4c, and 0x46. This translates to ‘ELF’ in ASCII, and is known as the Magic Number, used to identify ELF files. In fact, most files have a magic number, and can be identified by it. Windows exe files have the magic number 4D 5A, for example.
ELF files contain a few ‘headers’, which help us to read the ELF. The first of which, the File Header of length 64 bytes, ensures that data is correctly interpreted during linking or execution. Lots of useful information is contained in the File Header, including references to other headers, the file’s entry point, machine information, and even the Byte Order of the executable. To actually obtain this information though, we need to parse the File header as a sequence of bytes, and extract certain lengths of bytes based on whether the executable is 32 or 64 bit and the endianness. The structure of the ELF header is shown below in figure 3. Figure 2 outlines the fact that different ELF data-types have different purposes once read as bytes: they can be unsigned or signed values, and can serve different purposes depending on their specification.

5.2 Retrieving Function Names and Addresses

Once the ELF header has been parsed, of course we could proceed to disassemble an ELF file from the header's e_entry field. However, for a linear sweep, disassembly would just be of the entire executable code section, with no clear distinction between different functions, or extra useful symbolic information. Additionally, we would perhaps have to implement some (inaccurate) heuristic to discover functions not reachable from the entry point. Given that the ELF format provides means to accurately obtain function names and addresses within scope of the executable files, it makes sense to do so.

To obtain functions names, we must refer to data stored in the Section Header string table in the ELF. The Section Header table's offset is defined in the ELF header as e_shoff, with other additional useful fields e_shentsize (section header entry size), and e_shnum (number of section header entries). The Section Header table can then be accessed and parsed in the same was that the ELF header was parsed, to obtain information about different sections in the ELF. The sections we are concerned with in order to resolve function names by address are .strtab and .symtab.

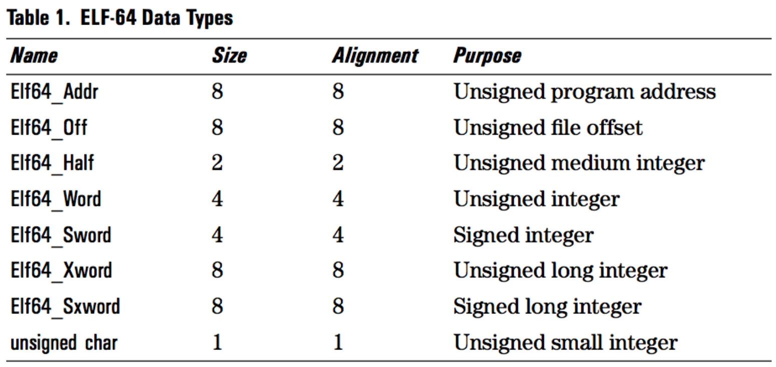
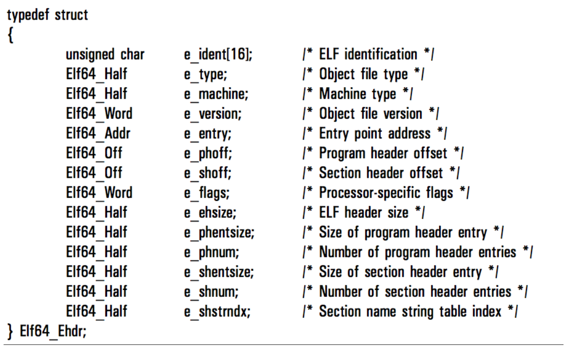


Table 1. ELF-64 Data Types

Name	*Size*	*Alignment*	*Purpose*
Elf64_Addr	8	8	Unsigned program address
Elf64_Off	8	8	Unsigned file offset
Elf64_Half	2	2	Unsigned medium integer
Elf64_Word	4	4	Unsigned integer
Elf64_Sword	4	4	Signed integer
Elf64_Xword	8	8	Unsigned long integer
Elf64_Sxword	8	8	Signed long integer
unsigned char	1	1	Unsigned small integer

Figure 2 – 64 bit ELF data types[6]



```
typedef struct
{
        unsigned char   e_ident[16];    /* ELF identification */
        Elf64_Half      e_type;         /* Object file type */
        Elf64_Half      e_machine;      /* Machine type */
        Elf64_Word      e_version;      /* Object file version */
        Elf64_Addr      e_entry;        /* Entry point address */
        Elf64_Off       e_phoff;        /* Program header offset */
        Elf64_Off       e_shoff;        /* Section header offset */
        Elf64_Word      e_flags;        /* Processor-specific flags */
        Elf64_Half      e_ehsize;       /* ELF header size */
        Elf64_Half      e_phentsize;    /* Size of program header entry */
        Elf64_Half      e_phnum;        /* Number of program header entries */
        Elf64_Half      e_shentsize;    /* Size of section header entry */
        Elf64_Half      e_shnum;        /* Number of section header entries */
        Elf64_Half      e_shstrndx;     /* Section name string table index */
} Elf64_Ehdr;
```

Figure 3 – ELF header section structure[5]

By parsing the Secion Header Table to obtain the offset, size, and entry size for each Section Header, we can obtain .symtab and .strtab by their name, and then proceed to obtaining function data.

.symtab entries have their own section structure, similar to that of Figure 3, but reserving a total of 24 bytes per entry. Every .symtab entry holds several pieces of information useful to us: st_name, st_value, st_info and st_size. st_size refers to the size of this entry in the symbol table, e.g. main could have a size of 220 bytes. The most significant four bits of st_info hold information about the linkage visibility of the symbol in question. The least significant four bits of st_info hold information about the type of symbol. If this has a value of '2', the symbol in question is a function, (a function is what we are mainly concerned with). st_value holds the actual absolute value or address of the symbol entry, crucially telling us where this symbol is located even in the file. If the address is absolute, we must analyse the LOAD segmenet of the ELF to determine how to interpret virtual addresses. This can be done by obtaining LOAD's physical address, and using it as a modifier to convert between virtual and physical addresses; subtracting the modifier from a virtual address to obtain a physical address and vice versa.

st_name holds the index at which the current symbol table entry's name is located in its string table (.strtab, as previously mentioned).

To finally get the name of a symbol, .strtab's st_name index must be accessed. Each byte at this index must be converted to an ASCII character until the null byte is reached, The Symbol name is the result of concatenating all of the ASCII characters read.

Of course, this method of retrieving function names within the ELF file is dependant on a few things. The most critical dependency is the existence of .symtab itself. For execution of ELF files, .symtab does not necessarily need to exist, due to the fact that it is non-allocable. That is, the section is only needed during link-time (when the ELF file is built by the 'linker') and not during the execution of the program. Inversely, the existence of the (allocable) .dynsym section is paramount to the execution of the program[7], containing required symbols for program execution. One could simply 'strip' the ELF, which discards all symbols from the file, apart from those stored in .dynsym. This would render this method for resolving function names useless. But in the case that the function name exists, it is sensible to resolve the function names this way.

Should it be noticed that the ELF is stripped, the .dynsym section can be parsed to extract symbols present there, but it does not offer as complete symbol coverage as parsing .symtab. Symbols in this table are also typically symbols representing shared libraries, but not always.

Chapter 6

Program Functionality

The final deliverable Disassembler program is a graphical user interface supporting a host of features. The goal is to present the CFG and associated disassembled instructions in a clear manner, such that a user can easily follow the flow of instructions in order to understand the input program's execution.

This section covers the functionality of the produced disassembler including its internal workings for some complex procedures.

6.1 User Interface

The disassembler's functionality is presented to the user through a graphical user interface. The objective of the user interface is to clearly display functions, a control flow graph, data visualisation, exporting of data, searching for instructions/addresses, and the exporting of a control flow graph.

Clearly presented to the user is a list of functions located in the executable file. Function names are resolved if possible, but simply have the function's start address set as the function name otherwise (such as for stripped ELFs). The disassembled instructions a presented in a table representing the set of all instructions disassembled. The table is searchable for any of its elements, which is quite convenient.

The control flow graph is slightly unconventional in that it allows call instructions to mark the end of a basic block, slightly overcomplicating the view of the graph. This is something that I realise and could fix, but at the cost of slight runtime when performing the disassembly, as blocks would need to be split or merged as seen in the solution to the problem of section **5.2.** This is ultimately just down to user preference and can always be implemented.

Data export was outlined in the original specification, and has been implemented such that a user can export the control flow graph, selecting the directory in which to save to. The user can also select and export instructions in a desired format, including raw hexadecimal representation, hexadecimal format with spaces, to a textual representation equivalent to that of the user interface table, showing the desired columns.

An additional feature of the program is to clearly present sections of the ELF to the user in order to show addresses of different sections. This can aid users to determine what section in the ELF that perhaps a referenced address is located in, since relocation are not made by the disassembler.

6.2 Locating Functions

Locating Functions in Stripped ELFs

For stripped ELF files, an experimental heuristic is applied to discover the main function. The program takes the entry point (the address of the _start function) and disassembles instructions up until a 'hlt' instruction is reached. When hlt is reached, the first call instruction before this is located, assumed to be a call to libc_start_main. libc_start_main takes as argument the memory address of the main function. This can either be pushed onto the stack directly or loaded into a register by a mov instruction or lea for example. The heuristic might fail in cases such as when start does not end with a hlt instruction, but has worked for all binaries tested in /bin and /usr/bin on my project's virtual OS, which are all stripped 64 bit ELF files.

The next step in determining all functions in stripped ELF files actually occurs after the recursive traversal has been performed. It is not as trivial as identifying function prologues, such as (push .bp) then (mov .bp, .sp) as these are not compulsory for compilers to include during assembly. Despite this, for programs with function prologues included, I have tested results and they have been great. However, compilers can simply be instructed to omit function prologues, so this is highly unsuitable. Instead, the way in which functions are located is by assuming that the destination of call instructions are the locations of functions, described below.

A Basic Block contains disassembly from a set start address up until the next branch instruction is reached, inclusive of this branch instruction (see 5.2 to understand Basic Blocks). Each block is assigned an 'Address Reference List', containing between zero and two addresses, being the addresses of the fall-through instruction (next instruction in sequence) and target of the block's branch instruction. Of course, a block may represent an end of a region, in which case it would have no references in its list. Additionally, a block may end with a normal instruction such as 'mov', in which case the address reference list would only contain the fall-through instruction.

The addresses in this list are known as 'children' of a Basic Block, and these blocks know their 'parent' blocks, after being set. The Address Reference List, crucially, does not include addresses of branch targets where the branch instruction is 'call'. Due to this property, addresses exclusively referenced by call instructions will <u>not</u> be in any block's address reference list. This means that Basic Blocks representing the target of call instructions, and thus functions, have <u>no</u> parent blocks. As an added feature dealing with a small corner case, the list of Basic Blocks maintained is iterated, dealing with blocks ending in a call instruction. For each block ending with a call instruction, the block referenced by the destination of the call is set to having *no* parents. The reason for this is that there were cases previously not taken into account whereby a jump instruction can have the destination address at an address also referenced by a call instruction, meaning that the block beginning with the destination has parents, which should not be a property of blocks referenced by call instructions.

We then iterate across the list of Basic Blocks, finding blocks with no parents and add them to a list of functions.

Locating Functions in Non-Stripped ELFs

For non-stripped ELF files, the addresses of the main function and all others are known before the disassembly algorithm executes, unlike for Stripped ELFs. The way in which functions are discovered for non-stripped ELFs is described in section **5.2**. This is done programmatically, however, working with the GNU Binutils Java library to first locate the addresses and size of the symbol and string tables. Once the program Symbol name and addresses have been resolved, each Symbol representing a function is added to a list of functions.

6.3 Disassembly Algorithm

The disassembly algorithm chosen for this project is the Recursive Traversal algorithm. I chose this algorithm for the advantages stated over the linear sweep algorithm in section **3.4.** Additionally, the recursive traversal algorithm is complementary to construction of the CFG.

In the code, a Basic Block is an object holding a sequence of instructions, each located directly after the previous instruction in the executable, with no branch instruction located at any point in the sequence apart from the very end. A Basic block may directly reference zero, one, or two other Basic Blocks due to the property that a branch instruction may only be located at the end of a block. Each Basic Block holds references to other blocks in a block's Address Reference list, with these known as child blocks. Basic Blocks with no parent blocks are said to be blocks representing the first instructions of a function.

Walkthrough

The algorithm starts by building the very first Basic Block, which represents the beginning of the main function. The first instruction located at the offset of main is disassembled and added to the block. The instruction is then subject to several checks. If the instruction is a ret instruction, the current block is returned and added to a list of known blocks. If the instruction is an unconditional or conditional control transfer instruction and the address can be directly resolved due to the fact that the operand is solely representative of a numerical value, proceed to deal with this instruction accordingly, by adding its jump target to a list of potential targets to disassemble at. Also, if the instruction is not a call instruction, its jump destination should be added to addresses referenced by the current block. If the operand does not represent a numerical value, the jump target is said to be to a value in some register, which it can't currently deal with.

Still within the scope of dealing with an conditional or unconditional jump instruction, a 'continue address' is then formulated as the (address currently being disassembled at) + (the current instruction size). This is effectively the .ip register value, which points to the next instruction to be executed at runtime.
The continue address is only added to the list of potential disassembly targets and reference addresses of the current block if the instruction is not an unconditional jump instruction, which always only continues disassembly at its jump target address, so has no fall-through. The current block is the returned and added to the list of known blocks.

After the above, we can see that if the instruction passed in was an unconditional / conditional control transfer instruction or a return instruction, the instruction has been dealt with and returns the current block; adding it to the list of known Basic Blocks.

In the event that the instruction wasn't any of these instructions, the address to disassemble at and therefore the next instruction in this block is set as the sum of the current address and size of the current instruction. The block is returned when a known address is reached.

The above procedure is carried out for every address in the potential disassembly targets list, and results in a list of known Basic Blocks being built.

6.4 Algorithm Weakness and Solution

A problem with the algorithm in section **5.2** is that a list of basic blocks has been constructed that may hold blocks which can (rarely) reference an address inside some other block. This arises due to the fact that possible disassembly targets are inserted at the tail of the possible disassembly target list as opposed to after the current index. The problem can be visualised in figure 5.
The below example shows how the problem is caused by the algorithm, where disassembly begins at 0x402C70. As an example of verbal description: 'at 2), a conditional jump is encountered, so the 0x402C70 block has been completed. Target of jump added to possible targets.'

e.g.

Jump target and fall-through added to targets, end of 0x402C700 block

1) Disassembling at 0x402C70…continuing — Possible targets:

2) 0x402CAC: je 0x402D50 encountered…. — Possible targets: 0x402D50, 0x402CB2

3) Disassembling at 0x402D50…continuing — Possible targets: 0x402CB2

Added to end of list

4) 0x402D56: jmp 0x402CB8 encountered… — Possible targets: 0x402CB2, 0x402CB8

Ideally jmp straight to 2CB8, disasming there, but 2CB8 inserted **after** 2CB2 instead

---

5) Disassembling at 0x402CB2…continuing — Possible targets: 0x402CB8

6) note: disassembled at 0x402CB8 — Possible targets: 0x402CB8

We just disassembled at 0x402CB8, as it was reached yet it's still a possible target of a block, as it is in the possible target list. This means that 0x402CB8 isn't rightfully treated as that beginning of a new Basic Block since it was just disassembled as a part of the block 0x402CB2. Therefore, the block containing jmp 0x402CB8 now references the block beginning 0x402CB2, so jumps mid-way into a block. See how a 0x402CB8 block is not created next….

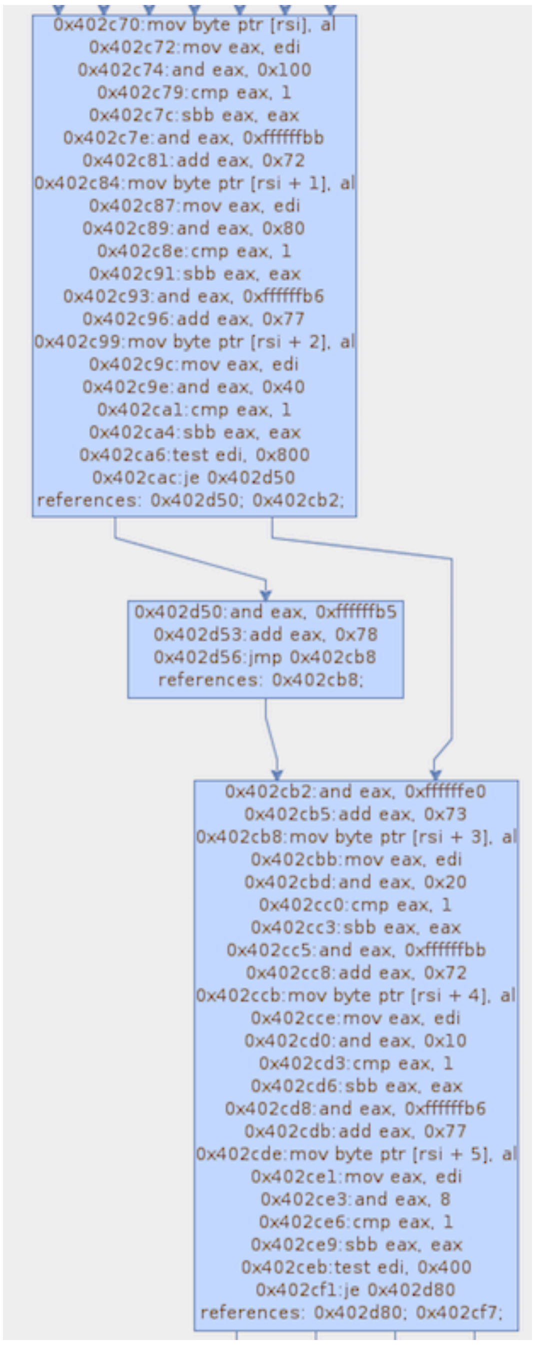
---

7) END OF 0x402CB2 block, ret reached — Possible targets: 0x402CB8

8) Disassembling at 0x402CB2…. — Possible targets:
8.1) ????????.....0x402CB2 already in visited address list…return current block

9) End of disassembly.

To overcome this problem, splitBlock() is invoked in the Disassemble class, splitting the problematic blocks such as the one beginning with 0x402CB2 in Figure 5 into blocks such as 0x402CB2 and 0x402CB8, as seen in Figure 6. The block with the jump to an instruction mid-way through it is patched such that it only contains instructions up until the mid-block instruction, and now references a newly created block containing instructions from the mid-block jump address onwards. The block referencing 0x402CB8 now happily reference a block beginning with that address, as it actually exists after the patch.

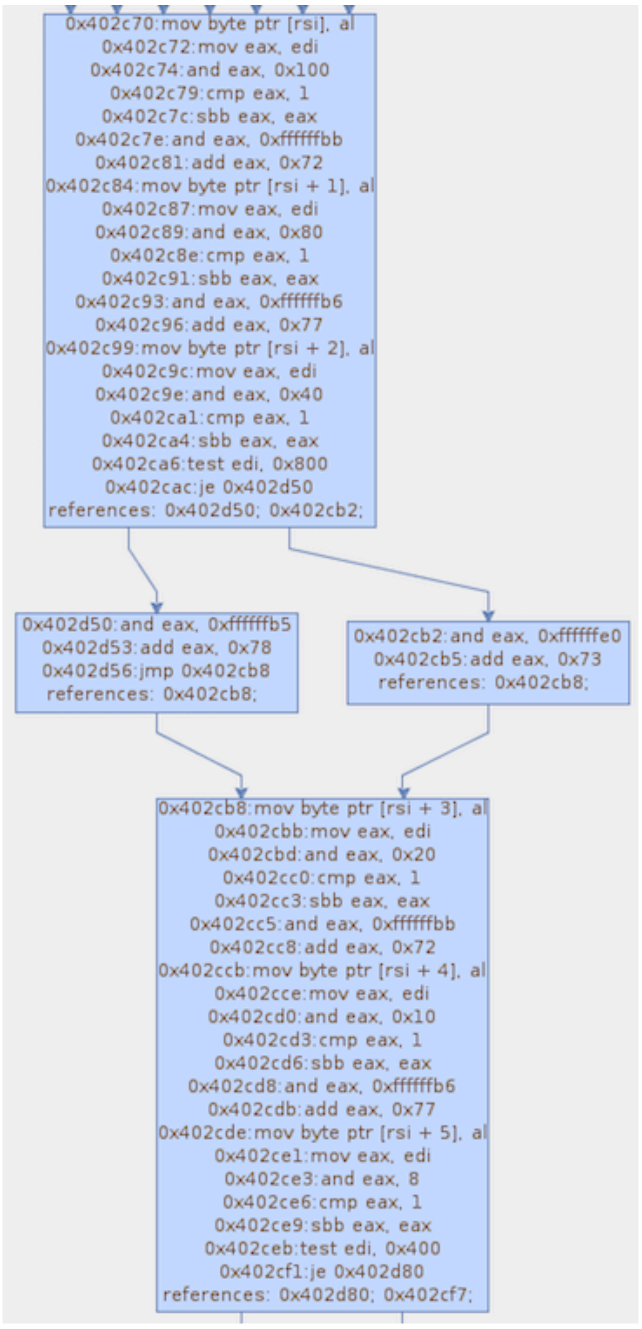


```
0x402c70:mov byte ptr [rsi], al
0x402c72:mov eax, edi
0x402c74:and eax, 0x100
0x402c79:cmp eax, 1
0x402c7c:sbb eax, eax
0x402c7e:and eax, 0xffffffbb
0x402c81:add eax, 0x72
0x402c84:mov byte ptr [rsi + 1], al
0x402c87:mov eax, edi
0x402c89:and eax, 0x80
0x402c8e:cmp eax, 1
0x402c91:sbb eax, eax
0x402c93:and eax, 0xffffffb6
0x402c96:add eax, 0x77
0x402c99:mov byte ptr [rsi + 2], al
0x402c9c:mov eax, edi
0x402c9e:and eax, 0x40
0x402ca1:cmp eax, 1
0x402ca4:sbb eax, eax
0x402ca6:test edi, 0x800
0x402cac:je 0x402d50
references: 0x402d50; 0x402cb2;

0x402d50:and eax, 0xffffffb5
0x402d53:add eax, 0x78
0x402d56:jmp 0x402cb8
references: 0x402cb8;

0x402cb2:and eax, 0xffffffe0
0x402cb5:add eax, 0x73
0x402cb8:mov byte ptr [rsi + 3], al
0x402cbb:mov eax, edi
0x402cbd:and eax, 0x20
0x402cc0:cmp eax, 1
0x402cc3:sbb eax, eax
0x402cc5:and eax, 0xffffffbb
0x402cc8:add eax, 0x72
0x402ccb:mov byte ptr [rsi + 4], al
0x402cce:mov eax, edi
0x402cd0:and eax, 0x10
0x402cd3:cmp eax, 1
0x402cd6:sbb eax, eax
0x402cd8:and eax, 0xffffffb6
0x402cdb:add eax, 0x77
0x402cde:mov byte ptr [rsi + 5], al
0x402ce1:mov eax, edi
0x402ce3:and eax, 8
0x402ce6:cmp eax, 1
0x402ce9:sbb eax, eax
0x402ceb:test edi, 0x400
0x402cf1:je 0x402d80
references: 0x402d80; 0x402cf7;
```

Figure 5 – result of mid-block reference



```
0x402c70:mov byte ptr [rsi], al
0x402c72:mov eax, edi
0x402c74:and eax, 0x100
0x402c79:cmp eax, 1
0x402c7c:sbb eax, eax
0x402c7e:and eax, 0xffffffbb
0x402c81:add eax, 0x72
0x402c84:mov byte ptr [rsi + 1], al
0x402c87:mov eax, edi
0x402c89:and eax, 0x80
0x402c8e:cmp eax, 1
0x402c91:sbb eax, eax
0x402c93:and eax, 0xffffffb6
0x402c96:add eax, 0x77
0x402c99:mov byte ptr [rsi + 2], al
0x402c9c:mov eax, edi
0x402c9e:and eax, 0x40
0x402ca1:cmp eax, 1
0x402ca4:sbb eax, eax
0x402ca6:test edi, 0x800
0x402cac:je 0x402d50
references: 0x402d50; 0x402cb2;

0x402d50:and eax, 0xffffffb5
0x402d53:add eax, 0x78
0x402d56:jmp 0x402cb8
references: 0x402cb8;

0x402cb2:and eax, 0xffffffe0
0x402cb5:add eax, 0x73
references: 0x402cb8;

0x402cb8:mov byte ptr [rsi + 3], al
0x402cbb:mov eax, edi
0x402cbd:and eax, 0x20
0x402cc0:cmp eax, 1
0x402cc3:sbb eax, eax
0x402cc5:and eax, 0xffffffbb
0x402cc8:add eax, 0x72
0x402ccb:mov byte ptr [rsi + 4], al
0x402cce:mov eax, edi
0x402cd0:and eax, 0x10
0x402cd3:cmp eax, 1
0x402cd6:sbb eax, eax
0x402cd8:and eax, 0xffffffb6
0x402cdb:add eax, 0x77
0x402cde:mov byte ptr [rsi + 5], al
0x402ce1:mov eax, edi
0x402ce3:and eax, 8
0x402ce6:cmp eax, 1
0x402ce9:sbb eax, eax
0x402ceb:test edi, 0x400
0x402cf1:je 0x402d80
references: 0x402d80; 0x402cf7;
```

Figure 6 – mid-block jump problem solved

The downside to the patch is that it introduced some added runtime complexity for the disassembly procedure, when there definitely exists a quicker fix. However, I feel more confident with this fix, given that it actually is proven to work, and that the issue was only noticed after the algorithm had been implemented. It should also be noted that instances such as in the error always involve all associated instructions being disassembled, so does not result in any erroneous or incorrect disassembly; just how we perceive Basic Blocks and the CFG.

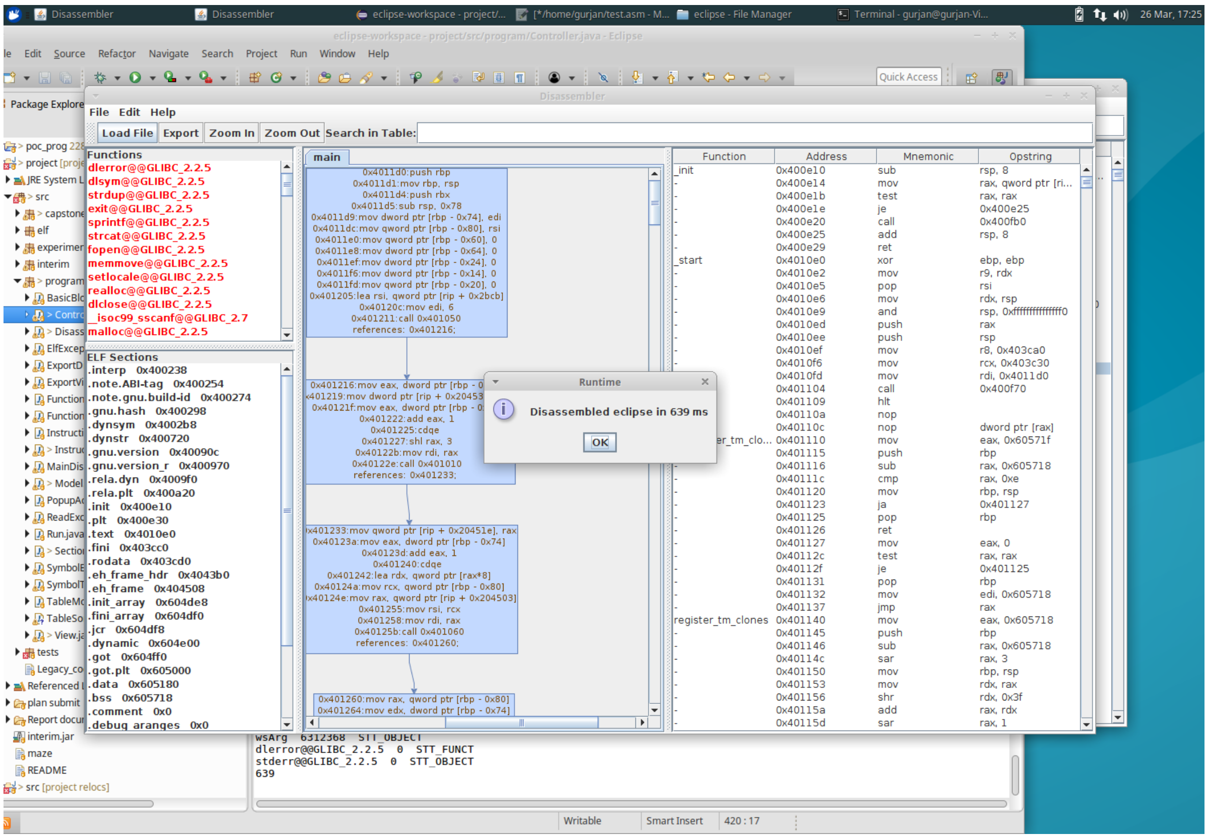
Chapter 7

Evaluation of Disassembler on Set Binaries

This section analyses the disassembler's operation, as well as mechanisms to test the output of disassembly, ensureing its correctness.

7.1 Runtime Performance

For a most basic runtime test, the disassembler was given input of the eclipse executable ELF. The binary is a non-stripped 64-bit ELF of size 71.5 kB. This was done in 639ms.



Functions

```
dlerror@@GLIBC_2.2.5
dlsym@@GLIBC_2.2.5
strdup@@GLIBC_2.2.5
exit@@GLIBC_2.2.5
sprintf@@GLIBC_2.2.5
strcat@@GLIBC_2.2.5
fopen@@GLIBC_2.2.5
memmove@@GLIBC_2.2.5
setlocale@@GLIBC_2.2.5
realloc@@GLIBC_2.2.5
dlclose@@GLIBC_2.2.5
__isoc99_sscanf@@GLIBC_2.7
malloc@@GLIBC_2.2.5
```

ELF Sections

```
.interp   0x400238
.note.ABI-tag   0x400254
.note.gnu.build-id   0x400274
.gnu.hash   0x400298
.dynsym   0x4002b8
.dynstr   0x400720
.gnu.version   0x40090c
.gnu.version_r   0x400970
.rela.dyn   0x4009f0
.rela.plt   0x400a20
.init   0x400e10
.plt   0x400e30
.text   0x4010e0
.fini   0x403cc0
.rodata   0x403cd0
.eh_frame_hdr   0x4043b0
.eh_frame   0x404508
.init_array   0x604de8
.fini_array   0x604df0
.jcr   0x604df8
.dynamic   0x604e00
.got   0x604ff0
.got.plt   0x605000
.data   0x605180
.bss   0x605718
.comment   0x0
.debug_aranges   0x0
```

```
0x4011d0:push rbp
0x4011d1:mov rbp, rsp
0x4011d4:push rbx
0x4011d5:sub rsp, 0x78
0x4011d9:mov dword ptr [rbp - 0x74], edi
0x4011dc:mov qword ptr [rbp - 0x80], rsi
0x4011e0:mov qword ptr [rbp - 0x60], 0
0x4011e8:mov dword ptr [rbp - 0x64], 0
0x4011ef:mov dword ptr [rbp - 0x24], 0
0x4011f6:mov dword ptr [rbp - 0x14], 0
0x4011fd:mov qword ptr [rbp - 0x20], 0
0x401205:lea rsi, qword ptr [rip + 0x2bcb]
0x40120c:mov edi, 6
0x401211:call 0x401050
references: 0x401216;

0x401216:mov eax, dword ptr [rbp - 0
x401219:mov dword ptr [rip + 0x20453
0x40121f:mov eax, dword ptr [rbp - 0
0x401222:add eax, 1
0x401225:cdqe
0x401227:shl rax, 3
0x40122b:mov rdi, rax
0x40122e:call 0x401010
references: 0x401233;

x401233:mov qword ptr [rip + 0x20451e], rax
0x40123a:mov eax, dword ptr [rbp - 0x74]
0x40123d:add eax, 1
0x401240:cdqe
0x401242:lea rdx, qword ptr [rax*8]
0x40124a:mov rcx, qword ptr [rbp - 0x80]
x40124e:mov rax, qword ptr [rip + 0x204503]
0x401255:mov rsi, rcx
0x401258:mov rdi, rax
0x40125b:call 0x401060
references: 0x401260;

0x401260:mov rax, qword ptr [rbp - 0x80]
0x401264:mov edx, dword ptr [rbp - 0x74]
```

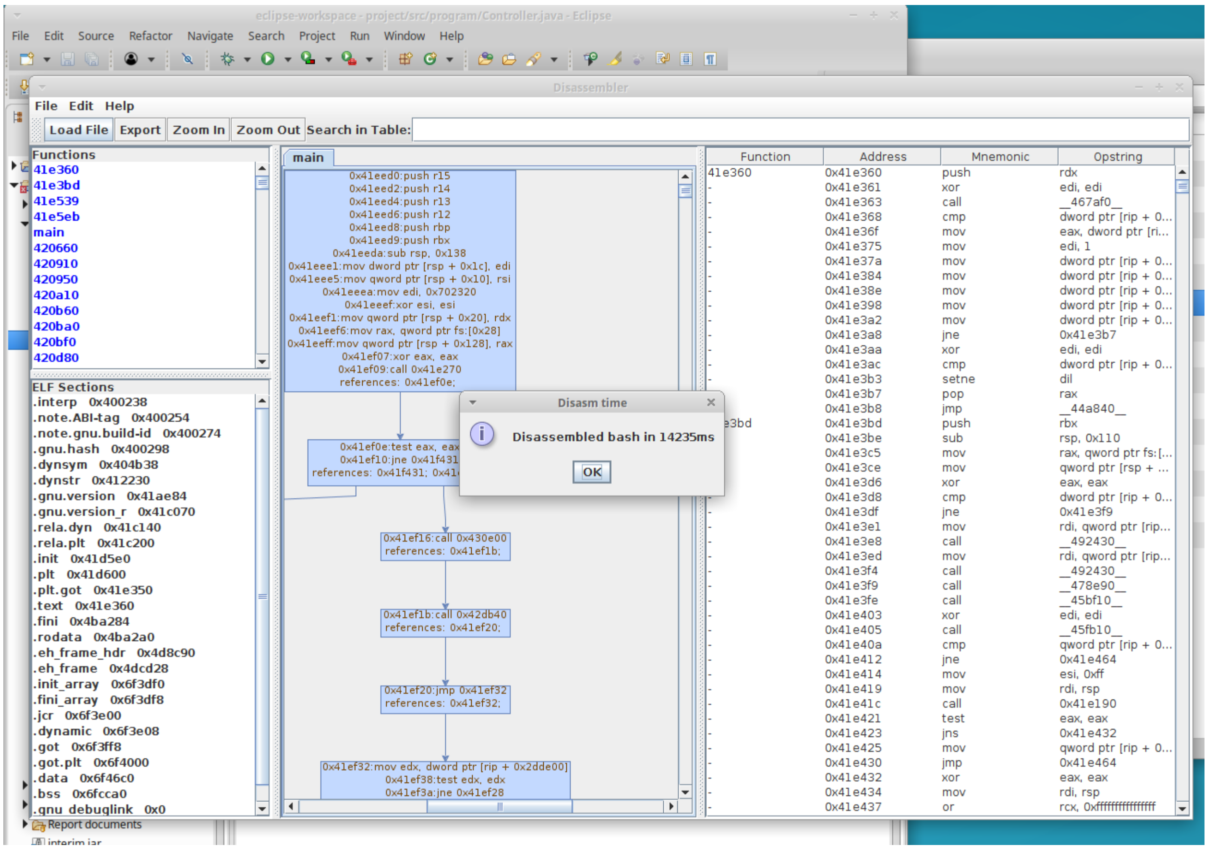
Disassembled eclipse in 639 ms

Function	Address	Mnemonic	Opstring
_init	0x400e10	sub	rsp, 8
-	0x400e14	mov	rax, qword ptr [ri...
-	0x400e1b	test	rax, rax
-	0x400e1e	je	0x400e25
-	0x400e20	call	0x400fb0
-	0x400e25	add	rsp, 8
-	0x400e29	ret	
_start	0x4010e0	xor	ebp, ebp
-	0x4010e2	mov	r9, rdx
-	0x4010e5	pop	rsi
-	0x4010e6	mov	rdx, rsp
-	0x4010e9	and	rsp, 0xfffffffffffffff0
-	0x4010ed	push	rax
-	0x4010ee	push	rsp
-	0x4010ef	mov	r8, 0x403ca0
-	0x4010f6	mov	rcx, 0x403c30
-	0x4010fd	mov	rdi, 0x4011d0
-	0x401104	call	0x400f70
-	0x401109	hlt	
-	0x40110a	nop	
-	0x40110c	nop	dword ptr [rax]
er_tm_clo...	0x401110	mov	eax, 0x60571f
-	0x401115	push	rbp
-	0x401116	sub	rax, 0x605718
-	0x40111c	cmp	rax, 0xe
-	0x401120	mov	rbp, rsp
-	0x401123	ja	0x401127
-	0x401125	pop	rbp
-	0x401126	ret	
-	0x401127	mov	eax, 0
-	0x40112c	test	rax, rax
-	0x40112f	je	0x401125
-	0x401131	pop	rbp
-	0x401132	mov	edi, 0x605718
-	0x401137	jmp	rax
register_tm_clones	0x401140	mov	eax, 0x605718
-	0x401145	push	rbp
-	0x401146	sub	rax, 0x605718
-	0x40114c	sar	rax, 3
-	0x401150	mov	rbp, rsp
-	0x401153	mov	rdx, rax
-	0x401156	shr	rdx, 0x3f
-	0x40115a	add	rax, rdx
-	0x40115d	sar	rax, 1

```
wsArg   6312368   STT_OBJECT
dlerror@@GLIBC_2.2.5   0   STT_FUNCT
stderr@@GLIBC_2.2.5   0   STT_OBJECT
639
```

Figure 7 – Disassembly time of Eclipse ELF

The disassembler can perform the analysis of the 1Mb linux ‘bash’ ELF in ~14 seconds, which is a decent result, I feel. The result is shown in Figure 8. I previously received results of ~58 seconds for the same executable. The disassembly procedure is very much open to being sped up, however, due to extra computations performed post-disassembly.



eclipse-workspace - project/src/program/Controller.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Disassembler
File Edit Help
Load File
Export
Zoom In
Zoom Out
Search in Table:
main
Disasm time
OK
Report documents
interim.jar

```
Functions
41e360
41e3bd
41e539
41e5eb
main
420660
420910
420950
420a10
420b60
420ba0
420bf0
420d80
```

```
ELF Sections
.interp   0x400238
.note.ABI-tag   0x400254
.note.gnu.build-id   0x400274
.gnu.hash   0x400298
.dynsym   0x404b38
.dynstr   0x412230
.gnu.version   0x41ae84
.gnu.version_r   0x41c070
.rela.dyn   0x41c140
.rela.plt   0x41c200
.init   0x41d5e0
.plt   0x41d600
.plt.got   0x41e350
.text   0x41e360
.fini   0x4ba284
.rodata   0x4ba2a0
.eh_frame_hdr   0x4d8c90
.eh_frame   0x4dcd28
.init_array   0x6f3df0
.fini_array   0x6f3df8
.jcr   0x6f3e00
.dynamic   0x6f3e08
.got   0x6f3ff8
.got.plt   0x6f4000
.data   0x6f46c0
.bss   0x6fcca0
.gnu_debuglink   0x0
```

```
0x41eed0:push r15
0x41eed2:push r14
0x41eed4:push r13
0x41eed6:push r12
0x41eed8:push rbp
0x41eed9:push rbx
0x41eeda:sub rsp, 0x138
0x41eee1:mov dword ptr [rsp + 0x1c], edi
0x41eee5:mov qword ptr [rsp + 0x10], rsi
0x41eeea:mov edi, 0x702320
0x41eeef:xor esi, esi
0x41eef1:mov qword ptr [rsp + 0x20], rdx
0x41eef6:mov rax, qword ptr fs:[0x28]
0x41eeff:mov qword ptr [rsp + 0x128], rax
0x41ef07:xor eax, eax
0x41ef09:call 0x41e270
references: 0x41ef0e;

0x41ef0e:test eax, eax
0x41ef10:jne 0x41f431
references: 0x41f431; 0x41

0x41ef16:call 0x430e00
references: 0x41ef1b;

0x41ef1b:call 0x42db40
references: 0x41ef20;

0x41ef20:jmp 0x41ef32
references: 0x41ef32;

0x41ef32:mov edx, dword ptr [rip + 0x2dde00]
0x41ef38:test edx, edx
0x41ef3a:jne 0x41ef28
```

```
Disassembled bash in 14235ms
```

Function	Address	Mnemonic	Opstring
41e360	0x41e360	push	rdx
-	0x41e361	xor	edi, edi
-	0x41e363	call	__467af0__
-	0x41e368	cmp	dword ptr [rip + 0...
-	0x41e36f	mov	eax, dword ptr [ri...
-	0x41e375	mov	edi, 1
-	0x41e37a	mov	dword ptr [rip + 0...
-	0x41e384	mov	dword ptr [rip + 0...
-	0x41e38e	mov	dword ptr [rip + 0...
-	0x41e398	mov	dword ptr [rip + 0...
-	0x41e3a2	mov	dword ptr [rip + 0...
-	0x41e3a8	jne	0x41e3b7
-	0x41e3aa	xor	edi, edi
-	0x41e3ac	cmp	dword ptr [rip + 0...
-	0x41e3b3	setne	dil
	0x41e3b7	pop	rax
	0x41e3b8	jmp	__44a840__
e3bd	0x41e3bd	push	rbx
	0x41e3be	sub	rsp, 0x110
	0x41e3c5	mov	rax, qword ptr fs:[...
	0x41e3ce	mov	qword ptr [rsp + ...
	0x41e3d6	xor	eax, eax
-	0x41e3d8	cmp	dword ptr [rip + 0...
-	0x41e3df	jne	0x41e3f9
-	0x41e3e1	mov	rdi, qword ptr [rip...
-	0x41e3e8	call	__492430__
-	0x41e3ed	mov	rdi, qword ptr [rip...
-	0x41e3f4	call	__492430__
-	0x41e3f9	call	__478e90__
-	0x41e3fe	call	__45bf10__
-	0x41e403	xor	edi, edi
-	0x41e405	call	__45fb10__
-	0x41e40a	cmp	qword ptr [rip + 0...
-	0x41e412	jne	0x41e464
-	0x41e414	mov	esi, 0xff
-	0x41e419	mov	rdi, rsp
-	0x41e41c	call	0x41e190
-	0x41e421	test	eax, eax
-	0x41e423	jns	0x41e432
-	0x41e425	mov	qword ptr [rip + 0...
-	0x41e430	jmp	0x41e464
-	0x41e432	xor	eax, eax
-	0x41e434	mov	rdi, rsp
-	0x41e437	or	rcx, 0xffffffffffffffff

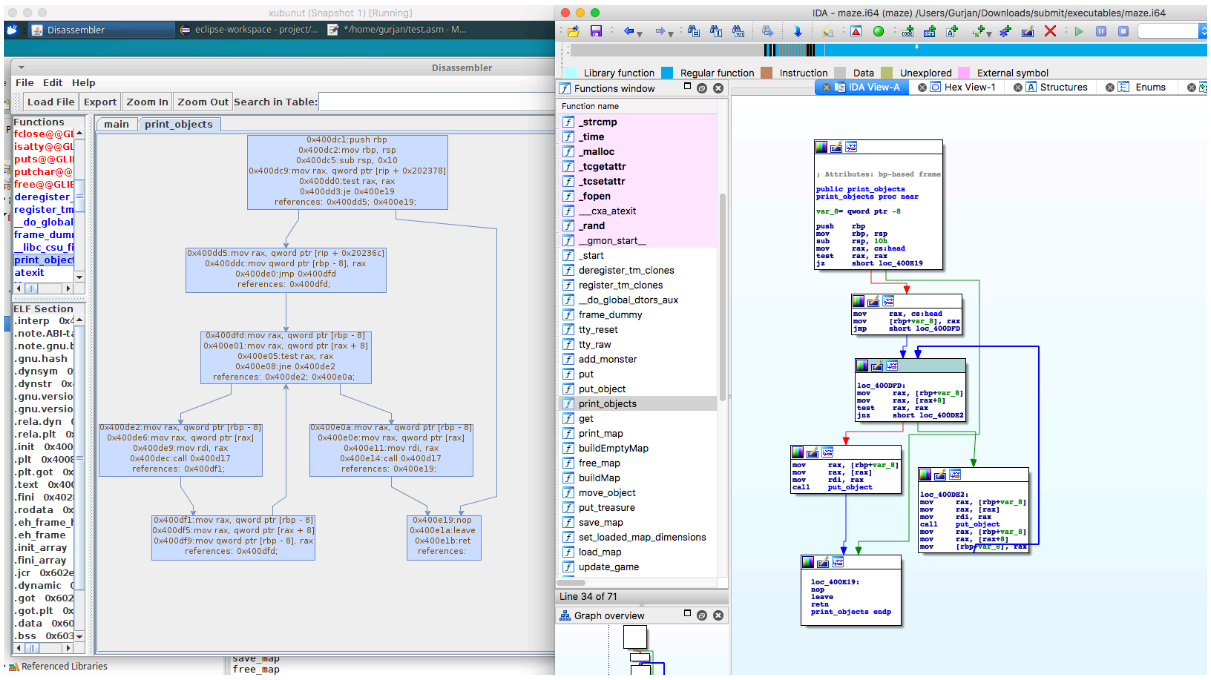
Figure 8 – Disassembly time of ‘bash’, ~14s

A faster algorithm constitutes construction of the control flow graph synchronously with the recursive traversal algorithm, in order to eliminate all post-disassembly computations related to control flow construction. The algorithm implemented only provides the data necessary to provide a control flow graph after instructions have been disassembled. Additionally, a more efficient algorithm and technically correct should not require the split block procedure or result in the issue seen in section **6.4**.

7.2 Testing Output

Simple Method of Testing CFG

A simple method of testing the control flow graph is to perform a side-by-side visual comparasion of the control flow graph produced by this disassembler, and the control flow graph produced by another disassembler. The other disassembler used was IDA Pro (free), for its reputation as the best available disassembler. The most basic view of this method of testing was to stand the output CFG up against a CFG produced by an industry-standard tool and analyse quality, similarities, and differences in the control flow graph. Figure 9 shows how this was done.



```
0x400dc1:push rbp
0x400dc2:mov rbp, rsp
0x400dc5:sub rsp, 0x10
0x400dc9:mov rax, qword ptr [rip + 0x202378]
0x400dd0:test rax, rax
0x400dd3:je 0x400e19
references: 0x400dd5; 0x400e19;

0x400dd5:mov rax, qword ptr [rip + 0x20236c]
0x400ddc:mov qword ptr [rbp - 8], rax
0x400de0:jmp 0x400dfd
references: 0x400dfd;

0x400dfd:mov rax, qword ptr [rbp - 8]
0x400e01:mov rax, qword ptr [rax + 8]
0x400e05:test rax, rax
0x400e08:jne 0x400de2
references: 0x400de2; 0x400e0a;

0x400de2:mov rax, qword ptr [rbp - 8]
0x400de6:mov rax, qword ptr [rax]
0x400de9:mov rdi, rax
0x400dec:call 0x400d17
references: 0x400df1;

0x400e0a:mov rax, qword ptr [rbp - 8]
0x400e0e:mov rax, qword ptr [rax]
0x400e11:mov rdi, rax
0x400e14:call 0x400d17
references: 0x400e19;

0x400df1:mov rax, qword ptr [rbp - 8]
0x400df5:mov rax, qword ptr [rax + 8]
0x400df9:mov qword ptr [rbp - 8], rax
references: 0x400dfd;

0x400e19:nop
0x400e1a:leave
0x400e1b:ret
references:
```

```
_strcmp
_time
_malloc
_tcgetattr
_tcsetattr
_fopen
__cxa_atexit
_rand
__gmon_start__
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
tty_reset
tty_raw
add_monster
put
put_object
print_objects
get
print_map
buildEmptyMap
free_map
buildMap
move_object
put_treasure
save_map
set_loaded_map_dimensions
load_map
update_game
```

```
; Attributes: bp-based frame

public print_objects
print_objects proc near

var_8= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     rax, cs:head
test    rax, rax
jz      short loc_400E19

mov     rax, cs:head
mov     [rbp+var_8], rax
jmp     short loc_400DFD

loc_400DFD:
mov     rax, [rbp+var_8]
mov     rax, [rax+8]
test    rax, rax
jnz     short loc_400DE2

mov     rax, [rbp+var_8]
mov     rax, [rax]
mov     rdi, rax
call    put_object

loc_400DE2:
mov     rax, [rbp+var_8]
mov     rax, [rax]
mov     rdi, rax
call    put_object
mov     rax, [rbp+var_8]
mov     rax, [rax+8]
mov     [rbp+var_8], rax

loc_400E19:
nop
leave
retn
print_objects endp
```

Figure 9 – Side-by-side comparison of Maze CFG in Disassembler and IDA

This is **not** a sound method of testing CFG output for larger binaries. It works for small functions and binaries as in Figure 9, as semantics can easily be compared, as well as disassembly code produced. However, there exist instances such as that shown by Figure 10 in which such as method of testing CFG output is virtually impossible.

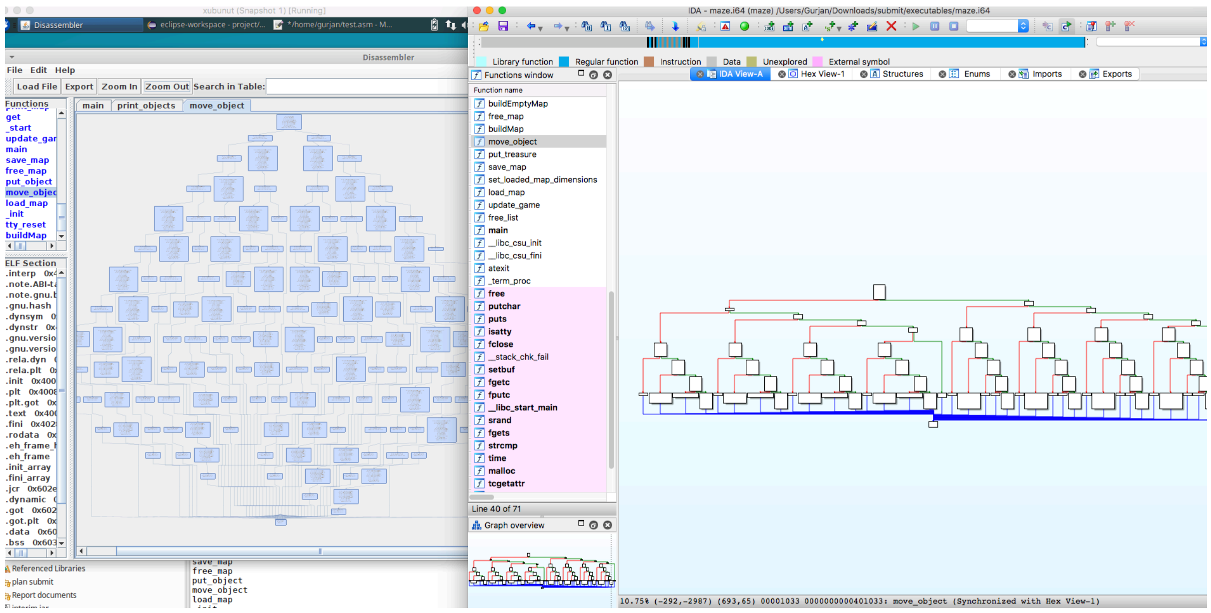


Figure 10 – The impossibility of comparing CFGs of the move_object function in Maze visually.

In these instances, we cannot easily visually compare the control flow graphs: the graph is extremely complicated. This disassembler's control flow graph also differs in that basic blocks can end in call instructions, meaning that the layout is different and there are naturally more blocks. Additionally, we may not even be able to trust IDA pro.

An Elaborate Testing Mechanism

BinDiff is a comparison tool for binary files, which assists vulnerability researchers and engineers to quickly find differences and similarities in disassembled code.[11] It is a powerful and incredibly user-friendly, useful tool, which has been used in the project as per the method described below. The tool takes as input two binaries and returns a visual representation of differences and similarities in the control flow graph and instructions. This means that in order to determine the difference between the original binary and a binary representing the disassembly instructions, the disassembled instructions must be reassembled. The tool does not operate directly on binaries, but on saved IDA instances, which means that two instances of IDA must be invoked with the binaries as input and the instances saved as an IDA Pro Database file. The IDA files can then be passed into BinDiff to obtain disparities between the binaries. Additionally, mutual function names are listed, as well as matching Basic Blocks identified.

For this project, BinDiff was used, but only for a small set of three test binaries. This is due to the fact that re-assembling the output for every instance of disassembly proved incredibly non-trivial for me. When re-assembling disassembly code, the code must be passed into an assembler. Assemblers used in the testing phase of this project were 'nasm' and 'GNU assembler'.

Such assemblers have certain syntax requirements that direct output from the disassembler do not obey, so disassembled instructions had to be passed into the assembler and then formatted accordingly. Along with this, by using readelf to find the variables in the .data section of the original executable, variables were declared in the .asm file. Additionally, disassembly output in the format given by Capstone can produce instructions such as 'dword ptr [….]'. The 'ptr' in this instruction is invalid, and will raise slightly ambiguous errors upon compilation. Once every syntactic issue had been resolved, the file could be assembled. Any other assembly errors encountered henceforth denoted that the output disassembly is incorrect, or executable otherwise. The results of comparing the binaries were then obtained from BinDiff, discussed next.

Observations
For all Basic Blocks matched during the testing process, the programmatic functionality of the blocks seemed to be the same, indicating correct disassembly for whichever blocks identified.

Functions traceable from the main function in the original executable are all present in the re-constructed executable for non-stripped ELFs. This was true for the input of the Maze executable, where all functions were presented, barring function calls outside the scope of the text section.

For stripped executables, it was observed that overall function discovery was much poorer, although still yielded decent results. The main issue is that only functions unreachable from the main function were disassembled, meaning that any other functions not directly called were not disassembled. Also, due to the function discovery mechanism being unable to deal with calls to register values, functions which are only referenced by calls to some value in a register are not discovered. This led to BinDiff raising differences between the executables, which was overwhelming at times for certain executables. However, it is reassuring to know the problems causing function discovery.

Testing Disassembly During Development

Development was highly testing orientated. For example, it had to be verified that Basic Blocks were correctly being created, matching the definition of a Basic Block. Each Basic Block was printed to the console and being analysed perspicaciously to ensure that results were in line with expected results. Such expected results included each block correctly referencing other blocks, referencing at most two other blocks, or ensuring that referenced addresses by a block were not in the middle of other blocks. The weakness discovered in section **6.4** was discovered in this manner. There were also issues related to the display of the control flow graph, whereby disassembly was correct, but the control flow graph had some minor errors due to its implementation programatically with the graph library.

Testing Obfuscation

The algorithm does not handle most obfuscation methods designed to thwart Recursive Traversal and, as such, obfuscated binaries were not provided as test input to the disassembler. However, results of disassembling binaries with certain obfuscation mechanisms were determined by manually running each step of the disassembly process on obfuscated areas by hand. These results have already been devised for obfuscation mechanisms mentioned in section **2.5** using this method, and result in the obfuscation succeeding in all instances.

Chapter 8

Software Engineering

This section outlines all aspects related to Software Engineering related to the project.

8.1 Initial Requirements

The project requirements were derived from the original project specification and were split into two Functional Requirements and Non-Functional Requirements.

Functional Requirements	
Determine entry point	Disassembled instruction visualisation
Determine Symbol names and addresses	Interaction with disassembled instructions
Determine file input type	Search disassembled instructions
Disassemble stripped ELFs	Control flow graph export
Perform Recursive Traversal	Instruction export
Determine function addresses from symbols	Input file structure visualisation
Determine functions heuristically/otherwise for stripped ELFs	Function identification from address
Show function names	Error handling
Control flow graph visualisation	Meaningful error messages
Interaction with CFG	

Figure 11 – Functional Requirements

In addition to these initial requirements, extra features were added during development to keep an open-minded approach to the project. Such features had to have been in line with the Non-Functional requirements though.

Non-Functional Requirements
Speed of disassembly - fast
Memory usage - low
Scalable for larger binaries
Reliability of results
Usability
Maintainable software
Testability of output
Source code readability

Figure 12 – Non-Functional Requirements

8.2 Methodology

An agile approach was taken to development of the software. This was the right approach to take due to the unpredictability associated with the production of this software. Theoretical aspects were constantly being learned during the project, even towards the very end, and an agile approach was absolutely necessary. As theory was learned throughout the course of the project, it was expected that concepts were discovered which could undermine previous assumptions made, meaning that the program would need to be adjusted accordingly. For example, the fact that function labels can be referenced by jump instructions (and not only call) was learned a long way into development. This meant that a solution had to be provided in the code, which patched previous functionality based on the assumption the functions could only be referenced by call instructions. The issue discovered in the code due to the theory learned was some *functions* not being discovered due to the fact that they were reference by a jump instruction, which assigned them a parent block (undermining the definition of start-of-function block having no parents). Situations like these were expected given that my initial knowledge of theoretical aspects related to the project was close to nothing, and were the reason that the Agile methodology was used.

The process of iterative development of program slices is defined by the Agile and was supplementary to the development of the project. Development began with printing all functions discovered to console. This in itself was a user story devised, and added value to the product. Thereafter, the recursive algorithm was implemented, and basic blocks could be printed to console, delimited by '------NEW BLOCK------' as an example. As soon as the recursive algorithm could do this, presentable disassembly had been obtained - a key deliverable component for the product developed. Functionality could then be built around this, for example showing instructions in a table, building a control flow graph, or data export. Each one of these components satisfied an appropriate user story, and added incremental value to the product as and when implemented.

The agile approach also complemented the production of working software, since the agile approach prioritises this over comprehensive documentation. This was especially important due to the experimental nature of the project and difficulty in ensuring that disassembly was correct, and coverage of executable instructions complete.

8.3 UML

A high level view of the program's *internal* functionality can be visualised with a UML activity diagram.

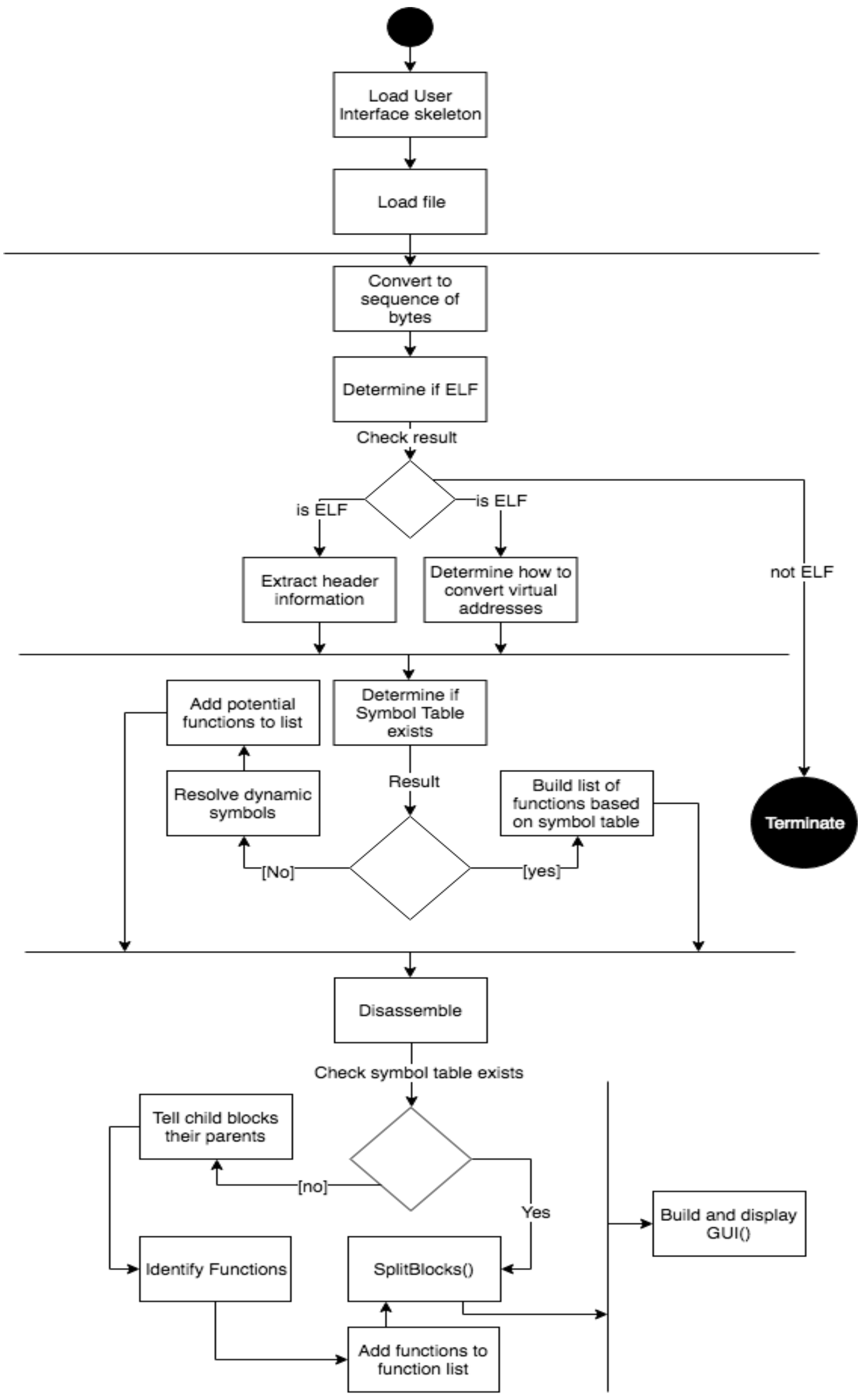

Load User Interface skeleton
Load file
Convert to sequence of bytes
Determine if ELF
Check result
is ELF
is ELF
not ELF
Extract header information
Determine how to convert virtual addresses
Terminate
Add potential functions to list
Determine if Symbol Table exists
Resolve dynamic symbols
Result
Build list of functions based on symbol table
[No]
[yes]
Disassemble
Check symbol table exists
Tell child blocks their parents
[no]
Yes
Build and display GUI()
Identify Functions
SplitBlocks()
Add functions to function list

Figure 13 – Activity Diagram of internal functionality

8.4 Architecture and Design Patterns

Architectural pattern: MVC
The MVC design pattern is incredibly useful for decoupling the disassembly Model and View, reducing design complexity. The MVC design pattern highly increases the Maintainability and Reusability of code. This is due to the fact that the disassembly Model is self-contained, meaning that adjustments can be easily made to it without the knowledge of the View and Controller. The Controller only cares about the output of data from the model being in a correct format such that it can be displayed in the View. We could potentially change the disassembly algorithm and, providing that it still returns a list of Basic Blocks which reference other blocks, all components of the user interface would be displayed as before.

MVC is implemented in the program for the main user interface through the classic use of a View, Controller, and Model. The View contains a list of fields local to it, representing all user interface components. When the View is invoked, its components are created through object declaration. Providing that the Model has performed its required computation, that is, the process of disassembly, the Controller is then invoked, filling the View's components with data provided by the Model. For example, the Controller uses Basic Block data from the model to display a CFG by analysing addresses associated with a function.

MVC is also implemented for the JDialogue box which appears when a set of instructions is exported. The ExportDialogue class represents the controller for this dialogue box, and the View class represented by ExportView, and uses the same Model as the main controller.

Design Pattern: Observer
The Observer pattern is used in the main controller and ExportDialogue controller in order to *observe* changes in the corresponding Views passed into them. Listeners initialised in each of the controllers observe changes in the View and respond accordingly. The use of this design pattern decouples classes, given that a View is not aware of any of its observers.

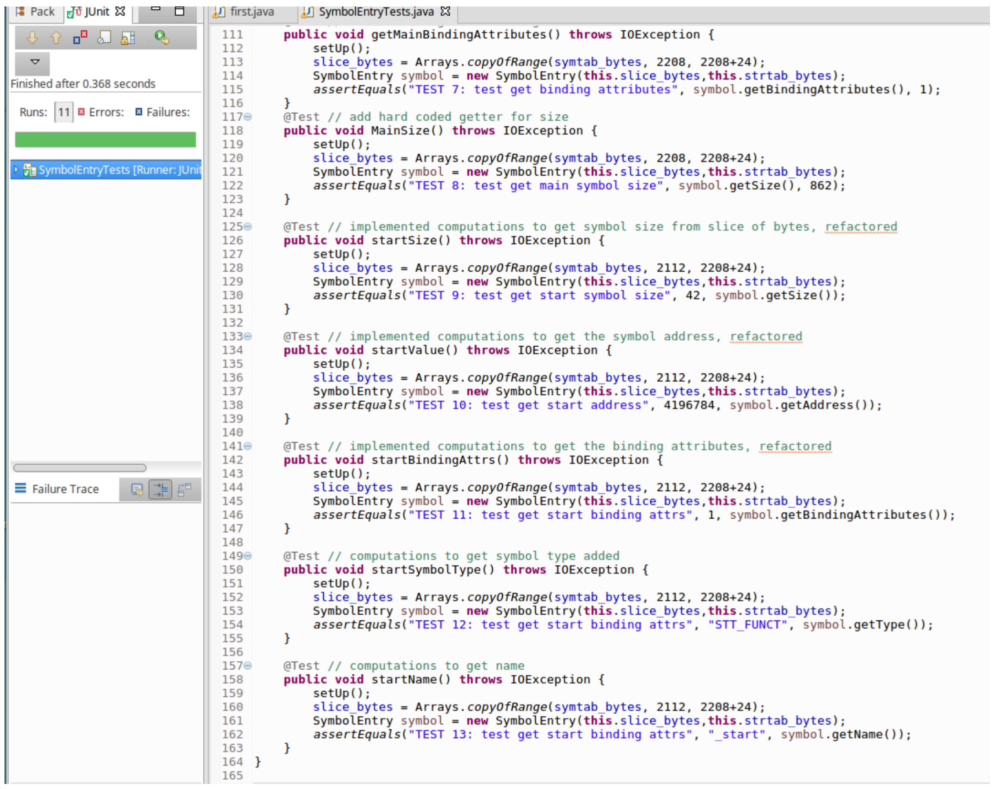
Design Pattern: Adapter
The Adapter pattern can be seen in the 'Model' class. Once the Controller discovers that a file has been selected in the view, the disassemble() method in Model is called from the Controller. This call instantiates an Object known as 'instance' (of class Disassemble) in the Model, with the file selected in the View as its argument. This invokes a sequence of methods and classes in the underlying subsystem, so that the disassembly procedure executes.

This pattern decouples the user interface from the complex workings of the subsystem performing disassembly.

Design Pattern: Singleton
The Disassemble class is a Singleton, which has a single instance at any time in the program, located in the Model. This is because we are only disassembling one file at a time, which means that there should only be one instance of disassembly.

8.5 Unit Testing

Some unit tests were conducted during development.
Below is a screenshot of several unit tests.



```java
    public void getMainBindingAttributes() throws IOException {
        setUp();
        slice_bytes = Arrays.copyOfRange(symtab_bytes, 2208, 2208+24);
        SymbolEntry symbol = new SymbolEntry(this.slice_bytes,this.strtab_bytes);
        assertEquals("TEST 7: test get binding attributes", symbol.getBindingAttributes(), 1);
    }
    @Test // add hard coded getter for size
    public void MainSize() throws IOException {
        setUp();
        slice_bytes = Arrays.copyOfRange(symtab_bytes, 2208, 2208+24);
        SymbolEntry symbol = new SymbolEntry(this.slice_bytes,this.strtab_bytes);
        assertEquals("TEST 8: test get main symbol size", symbol.getSize(), 862);
    }

    @Test // implemented computations to get symbol size from slice of bytes, refactored
    public void startSize() throws IOException {
        setUp();
        slice_bytes = Arrays.copyOfRange(symtab_bytes, 2112, 2208+24);
        SymbolEntry symbol = new SymbolEntry(this.slice_bytes,this.strtab_bytes);
        assertEquals("TEST 9: test get start symbol size", 42, symbol.getSize());
    }

    @Test // implemented computations to get the symbol address, refactored
    public void startValue() throws IOException {
        setUp();
        slice_bytes = Arrays.copyOfRange(symtab_bytes, 2112, 2208+24);
        SymbolEntry symbol = new SymbolEntry(this.slice_bytes,this.strtab_bytes);
        assertEquals("TEST 10: test get start address", 4196784, symbol.getAddress());
    }

    @Test // implemented computations to get the binding attributes, refactored
    public void startBindingAttrs() throws IOException {
        setUp();
        slice_bytes = Arrays.copyOfRange(symtab_bytes, 2112, 2208+24);
        SymbolEntry symbol = new SymbolEntry(this.slice_bytes,this.strtab_bytes);
        assertEquals("TEST 11: test get start binding attrs", 1, symbol.getBindingAttributes());
    }

    @Test // computations to get symbol type added
    public void startSymbolType() throws IOException {
        setUp();
        slice_bytes = Arrays.copyOfRange(symtab_bytes, 2112, 2208+24);
        SymbolEntry symbol = new SymbolEntry(this.slice_bytes,this.strtab_bytes);
        assertEquals("TEST 12: test get start binding attrs", "STT_FUNCT", symbol.getType());
    }

    @Test // computations to get name
    public void startName() throws IOException {
        setUp();
        slice_bytes = Arrays.copyOfRange(symtab_bytes, 2112, 2208+24);
        SymbolEntry symbol = new SymbolEntry(this.slice_bytes,this.strtab_bytes);
        assertEquals("TEST 13: test get start binding attrs", "_start", symbol.getName());
    }
}
```

Figure 14 – unit tests

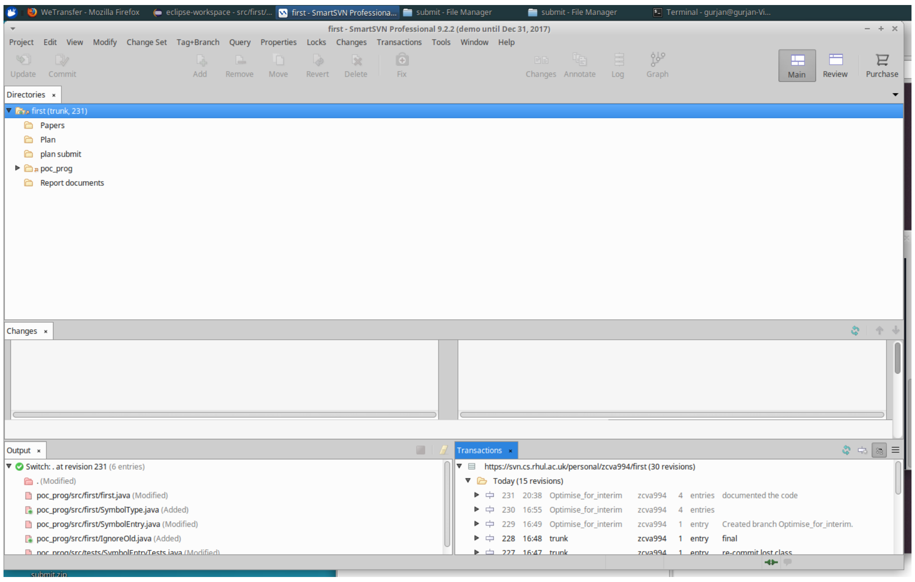
Methods in the SymbolEntry class were developed, such that each public method created was due to a unit test being performed. It is of course difficult to unit test for parsing an executable, as these unit tests do, as we do not necessarily know what certain value are expected to be, but several commands on linux were able to show the information required as expected values for tests.

The tests for the SymbolEntry class were done such that information from the symbol table representating the main function was used as expected values for the tests. With the expected results defined, the method of obtaining the results were initially done such that the test failed when first written, and always passed after adding just enough code to do so. Tests were then executed for the _start function, but with start's expected results for each test. Previous tests would ultimately fail, due to the hard coded test return values. Enough code was produced to then ensure that each test passed, as expected. For every test that passed, related code was refactored.

Similar tests were performed for pattern matching in the matchingTests test class.

8.6 Repository Usage

The project up until the interim view was maintained in SVN, but I really was not compatible with SVN, as specified in the interim report, so I ported the project over into a new GitHub repository for implementation of the main program.


WeTransfer - Mozilla Firefox
eclipse-workspace - src/first/...
first - SmartSVN Professiona...
submit - File Manager
submit - File Manager
Terminal - gurjan@gurjan-Vi...
first - SmartSVN Professional 9.2.2 (demo until Dec 31, 2017)
Project Edit View Modify Change Set Tag+Branch Query Properties Locks Changes Transactions Tools Window Help
Update Commit Add Remove Move Revert Delete Fix Changes Annotate Log Graph Main Review Purchase
Directories
Changes
Output
Transactions
submit.zip

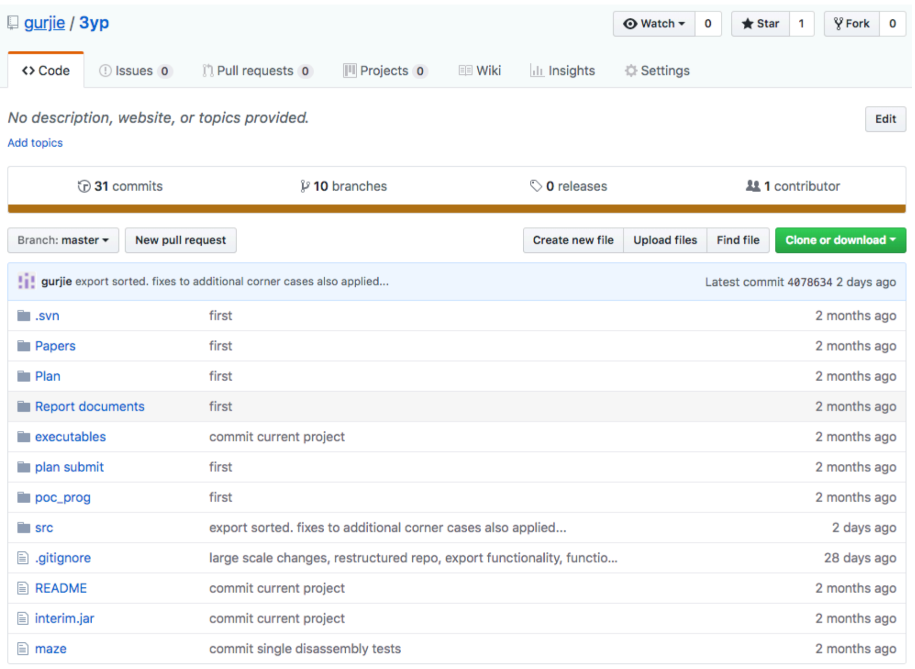
```
▼ first (trunk, 231)
    Papers
    Plan
    plan submit
  ▶ poc_prog
    Report documents
```

```
▼ Switch: . at revision 231 (6 entries)
    . (Modified)
    poc_prog/src/first/first.java (Modified)
    poc_prog/src/first/SymbolType.java (Added)
    poc_prog/src/first/SymbolEntry.java (Modified)
    poc_prog/src/first/IgnoreOld.java (Added)
    poc_prog/src/tests/SymbolEntryTests.java (Modified)
```

```
▼ https://svn.cs.rhul.ac.uk/personal/zcva994/first (30 revisions)
  ▼ Today (15 revisions)
    ▶ 231  20:38  Optimise_for_interim  zcva994  4  entries  documented the code
    ▶ 230  16:55  Optimise_for_interim  zcva994  4  entries
    ▶ 229  16:49  Optimise_for_interim  zcva994  1  entry    Created branch Optimise_for_interim.
    ▶ 228  16:48  trunk                 zcva994  1  entry    final
    ▶ 227  16:47  trunk                 zcva994  1  entry    re-commit lost class
```

Figure 15 – Original SVN repository

The current repository can be found on GitHub at https://github.com/gurjie/3yp



No description, website, or topics provided.

31 commits | 10 branches | 0 releases | 1 contributor

gurjie export sorted. fixes to additional corner cases also applied... Latest commit 4078634 2 days ago

.svn	first	2 months ago
Papers	first	2 months ago
Plan	first	2 months ago
Report documents	first	2 months ago
executables	commit current project	2 months ago
plan submit	first	2 months ago
poc_prog	first	2 months ago
src	export sorted. fixes to additional corner cases also applied...	2 days ago
.gitignore	large scale changes, restructured repo, export functionality, functio...	28 days ago
README	commit current project	2 months ago
interim.jar	commit current project	2 months ago
maze	commit single disassembly tests	2 months ago

Figure 16 – GitHub repository for final program

The repository features in excess of 10 branches and 30 commits including all related project documents and executables.

8.7 Development Environment

A specialised development environment had to be setup in order to pursue this project. Initially, I was working on a Mac and had installed the Capstone Engine in order to understand its functionality. I quickly learned, however, that in order to disassemble ELF files, I needed to be working in a Linux environment to be able to quickly compile ELFs. This led me to installing VirtualBox and working in a virtualised environment.

After installing and testing several environments, I decided to use Xubuntu due to it being light-weight in nature and stable. A lot of issues were encountered, but it was finally setup with Eclipse, smartsvn, the Capstone Library installed and other software development dependencies. The JNA native library also had to be installed so that Capstone bindings would work.

Some way into development, several other Java graph libraries were also installed in order to find the library most suitable to represent a control flow graph.

Chapter 9

Self-Analysis

This section outlines Overall Achievements, a Reflection of Personal Progress, Future goals, and a Conclusion.

9.1 Overall Achievements

I am happy with the achievements of the project overall. While there is obvious room for improvement, all basic requirements from the specification have been met, each to some extent better than others. The algorithm used is the recursive traversal algorithm, which is a little more non-trivial than the linear sweep algorithm. I am happy with this, since the majority of powerful disassemblers implement this algorithm, so the disassembler's internal functionality could be considered to be of a high level. The disassembler can display a control flow graph clearly, and has support for data export. File parsing has been implemented to determine basic information related to executables passed in as input, such as the entry point, symbol table, the executable's virtual address conversion mechanism, and more. Earlier versions of the program parsed the .rela.plt table, but functional requirements were slightly modified such that this was made redundant. The piece of software produced also successfully disassembles stripped and unstripped ELF files.

I feel to have an understanding of how to build a user interface in Java. This is something prior to the project which I had always been interested in doing. I had great difficulty implementing some of the components in the user interface, and it feels good to have finally overcome these issues.

In addition to the software product produced, lots of theory has been learned, related the disassembly process, as well as Reverse Engineering. Coming into this project, my theoretical knowledge of the two concepts was close to nothing. I have also gained more confidence in my programming and Software Engineering skills, as well as a lower level understanding of how machines execute code. I could not be happier with the amount learned during the project about fundamental concepts related to computer science.

I feel that I have come a long way from assuming that executable files could simply be converted to a stream of bytes and passed into Capstone Engine's disasm() function in order to disassemble an executable! I have also gone from an understanding of absolutely nothing related to executable files (quite literally, thinking that they were all raw binary files which could be executed on any machine), to quite a bit of knowledge now.

9.2 Possible Improvements

I feel that I could have done more with this project, but I will re-state that I cannot be happier with the progress that have made overall. Aside from previously mentioned improvements to the disassembly algorithm and runtime performance, there is definitely more potential functionality to be added.

Some functionality would be quite simple:

- Disassemble different file formats
- Perform relocations in the ELF to programmatically in order obtain function names called and addresses in the procedural linkage table referenced in disassembly

The reason that relocations were not implemented in the final product was that it reached an unrealistic time in the project for me to add this feature by the time the theory behind doing so learned. I would definitely extend the program functionality to do this, however, in order to provide a better overall user experience, such that the user can view when and which calls to shared libraries are made. Disassembling different file formats would also be fairly simple, since learning the ELF has provided a great foundation for which to learn the layout of other executable files. I feel that if I have learned how to interpret ELF files, I can definitely do the same for other types of executable.

A great added functionality discussed with Dr Lorenzo Cavallaro of Royal Holloway's Information Security group would be the ability for the disassembler to perform what is known as 'value set analysis'. The concept, outlined in [18] in order to approximate the values of registers on the fly. VSA is a static analysis technique, combining numerical and pointer analysis. The disassembler does not implement any analysis of the value of registers, apart from in the heuristic discovering start if a register is loaded the with a value prior to calling libc_start_main.

9.3 Conclusion

This project has been one of the best learning experiences of my life, and I am happy to have chosen it. Equally, material learned along the way, as well as, during development has been invaluable. I will definitely try to extend the disassembler to achieve the possible imporvements in section **9.2** in the future.

Chapter 10

Appendices

This section outlines Overall Achievements, a Reflection of Personal Progress, Future goals, and a Conclusion.

10.1 Diary

24th September 2017	General acquintance with disassembly & biggest challenge realised: you can't separate data from code. Entirely alien concepts covered…seriously…
1st October 2017	Exploring concepts of Linear Sweep and Recursive Traversal Problems touched upon including jumping back to middle of an instruction.
8th October 2017	What exactly is good disassembly? Is there perfect disassembly? What can thwart disassembly? Obfuscation techniques touched upon
15th October 2017	Instruction sets, the actual process of disassembly itself, decoding instructions, Intel System ABI covered somewhat, in depth understanding attempted.
22th October 2017	Exploration of current disassemblers, their features, what they're doing, and how they do it. Looking at control flow analysis, trying to make sense of output lined up against input.
29th October 2017	How do we acutally begin to disassemble a file? Backgorund theory discovered in enough detail to begin. Began by try to disassembly with Capstone in Java, straight away.
5rd November 2017	After trying to make sense of things, the disassembler was simply a mess. Trying to disassemble Mach-O fiels from their 0th bye straight into an array of bytes and into Capstone. Concept of executable files themselves being by and large important. Realization that some development environment has to be setup. Tried to work with several Linux distro's, frustrating days, given most weren't suitable + software errors…
12th Novemeber 2017	Begging to explore the concept of file parsing, and manually reading executable files. Searched for any alternative to this and found a compromise solution. Solid understanding at this point of how file parsing worked, and started to feel confident about the project for the first time.
19th November 2017	Initial Version of the Disassembler written, output verification proceeded. This was the single most valuable week so far, where everything learnt beforehand suddenly just seemed to make sense. Linux tools assistang the development and verifying output discovered, working with readelf, objdump etc.

26th November 2017	Reading further into theory. Knowledge starting to really get strong at this point. Finding enjoyment in talking about disassembly and surrounding concepts to other people! Things like On-Demand disassembly discovered, all sorts of unique interpretations of the Linear sweep and Recursive traversal. Report writing finalised, combined with reports drafted throughough the months so far.
December 2017	Read plenty of papers covering different types of mechanisms to deal with obfuscation. Trying to understand which algorithm to implement.
January 1st 2018	Learned more about obfuscation by this stage, also started reading up on theory related to symbol table not displaying function addresses for shared libraries as expected.
January 8th 2018	Obfuscation methods finally understood, but now trying to understand algorithms related. In particular, the paper [10] was discovered and an ambitious attempt made to understand it.
Jenuary 15th 2018	Algorithm devised to deal with obfuscation mechanisms in [10]. Realising that implementation will not be trivial, and questioning whether this will actually be possible. Learned about function discover heuristics.
January 23rd 2018	Learned that the project requires only a recursive traversal algorithm. No need for the complex algorithm in [10]. Slightly relived, but also upset at wasted time.
January 30th 2018	GitHub repository setip. Initial commit
January 31st 2018	First disassembly programatic procedures of the ginal project begun. This involved working out how to perform the disassembly of single instructions. First graph library tested, not too sure how it will actually work – very skeptical.
February 11th – 17th 2018	Researching potential libraries to implement a control flow graph and how to actually do this from a programming pint of view. Libraries seem tricky to use, and can't seem to run a lot of libraries properly. Experimentation with Maven, never used it before but looks quite good. Decided not to use Maven. Devising graphical user interface designs.
February 21st 2018	Control flow graph library decided upon. First stage of implementation, with the MVC pattern in mind. Classes designed as such. 21st February 2018. GUI skeleton
February 22nd 2018	MVC and CFG fuunctionality furthered
February 23rd 2018	Changes to GUI to enable flexibility with regards to potential future development in the future. Test the control flow graph library with sample nodes and edges
February 24th 2018	Control flow graph can in theory now be built. Test cases built for basic block class to mark the first stage of back-end development. File loading added to functionality, disassembly model adapter devised and skeleton implemented, so that MVC can be fully implemented. Exception handling implemented.
February 27th 2018	Export functionality added for the control flow graph, although control flow graph doesn't work
February 28th 2018	Still unsure about exact algorithm structure, working around the clock to decide on a decent algorithm. It has mostly been devised, but needs more work.
March 9th 2018	Disassembly algorithm implemented. Bugs do exist though.

	Bugs have to be washed out. Validity of results to proceed.
March 10th 2018	Algorithm works properly. Fucntionality for functions still required…
March 11th 2018	CFG displays for functions, requires cleaning up, but can disply control flow graph elements related to a function.
March 12th 2018	CFG in line with IDA. Testing phase begins
March 14th 2018	Issues discovered during testing of disassembly and the control flow graph. Modify the algorithm to fix the problems….
March 16th 2018	Bug fixes, CFG should be correct, verify output again
March 17th 2018	Issues with disassembly and CFG again according to IDA and BinDiff. Find a patch. CFG can also be shown in tabs in the UI such that control flow graphs can be swithed between
March 18th 2018	Stripped ELFs can be disassembled in some cases, heuristics added to discover main, but they are vastly unreliable
March 19th 2018	Instruction table added to the user interface. The functionality is satisfactory, displaying functions but more is required for it to be a succesful user interface
March 20th 2018	CFG bug discovered, fix ported to program – needs testing. Perform testing
March 21st 2018	Export functionality complete to get instructions from the UI via export or copy/paste
March 26th 2018	Issues related to export sorted. Corner case fixes applied, yielding solid disassembly results.
March 27th 2018	Concrete disassembly results, perfect for the capabilities of the algorithm with regards to function discovery and code in the scope of the algorithm

10.2 User Manual

10.2.1 Installation

To install the disassembler, it is required that the Capstone shared library is installed on your computer.
See:

- https://www.capstone-engine.org/download.html for information on how to download Capstone
- https://www.capstone-engine.org/documentation.html for information on how to install Capstone

Download and installation instructions are very simple, with many different methods of installation for a whole host of operating systems.

Due to unexplainable issues building a .jar file in my virtual environment through Eclipse, the way in which the program can be run is through **Eclipse** and installation will be explained using eclipse.

1) Open Eclipse

2) Go to File → Open Project from File system → Locate and select the submitted 'project' folder

3) Under 'folders' in the import project window, select the src folder with path 'project → src' and then select finish.

4) We now need to set JARs used on the build path. Select the project foler and navigate to the 'Project' option on the eclispe toolbar.

5) Select 'Properties'

6) under 'Libraries', click Add External Jars and then add the two JARs shown in the figure below.

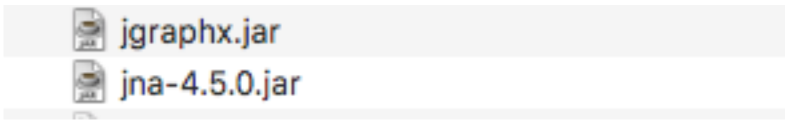

jgraphx.jar
jna-4.5.0.jar

Figure 17 – JARs to add to build path

7) The disassembler can now be run in Eclispe.

Apologies for having to run the project from source code, but I couldn't figure out how to export the JAR how it was exported in the interim project, as ambigous errors were raised.

10.2.2 Disassembling and General Usage

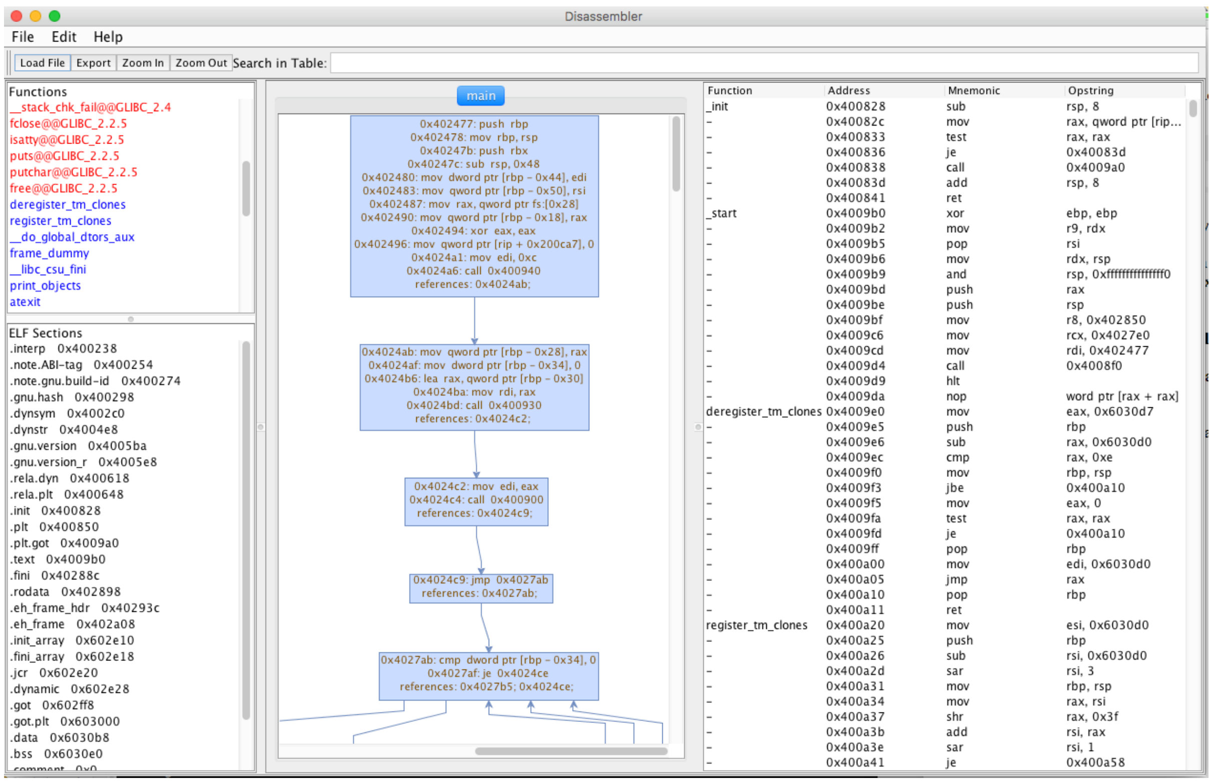
1) Select File → load executable and then open the file you wish to disassemble.
Alternatively, select Load File in the toolbar

NOTE: Some executables have been provided in the submitted folder; /executables

2) You may be notified if an executable is stripped. Press okay

3) Figure 18 shows Functions disassembled. Functions are red if their addresses could not be determined; shared libraries. Blue functions have been successfully disassembled

4) Very clearly displayed to you is the CFG, instruction table, function list, and a seciton showing ELF sections and their offsets



Functions
__stack_chk_fail@@GLIBC_2.4
fclose@@GLIBC_2.2.5
isatty@@GLIBC_2.2.5
puts@@GLIBC_2.2.5
putchar@@GLIBC_2.2.5
free@@GLIBC_2.2.5
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
__libc_csu_fini
print_objects
atexit

ELF Sections	
.interp	0x400238
.note.ABI-tag	0x400254
.note.gnu.build-id	0x400274
.gnu.hash	0x400298
.dynsym	0x4002c0
.dynstr	0x4004e8
.gnu.version	0x4005ba
.gnu.version_r	0x4005e8
.rela.dyn	0x400618
.rela.plt	0x400648
.init	0x400828
.plt	0x400850
.plt.got	0x4009a0
.text	0x4009b0
.fini	0x40288c
.rodata	0x402898
.eh_frame_hdr	0x40293c
.eh_frame	0x402a08
.init_array	0x602e10
.fini_array	0x602e18
.jcr	0x602e20
.dynamic	0x602e28
.got	0x602ff8
.got.plt	0x603000
.data	0x6030b8
.bss	0x6030e0
.comment	0x0

```
0x402477: push  rbp
0x402478: mov  rbp, rsp
0x40247b: push  rbx
0x40247c: sub  rsp, 0x48
0x402480: mov  dword ptr [rbp - 0x44], edi
0x402483: mov  qword ptr [rbp - 0x50], rsi
0x402487: mov  rax, qword ptr fs:[0x28]
0x402490: mov  qword ptr [rbp - 0x18], rax
0x402494: xor  eax, eax
0x402496: mov  qword ptr [rip + 0x200ca7], 0
0x4024a1: mov  edi, 0xc
0x4024a6: call  0x400940
references: 0x4024ab;

0x4024ab: mov  qword ptr [rbp - 0x28], rax
0x4024af: mov  dword ptr [rbp - 0x34], 0
0x4024b6: lea  rax, qword ptr [rbp - 0x30]
0x4024ba: mov  rdi, rax
0x4024bd: call  0x400930
references: 0x4024c2;

0x4024c2: mov  edi, eax
0x4024c4: call  0x400900
references: 0x4024c9;

0x4024c9: jmp  0x4027ab
references: 0x4027ab;

0x4027ab: cmp  dword ptr [rbp - 0x34], 0
0x4027af: je  0x4024ce
references: 0x4027b5; 0x4024ce;
```

Function	Address	Mnemonic	Opstring
_init	0x400828	sub	rsp, 8
–	0x40082c	mov	rax, qword ptr [rip...
–	0x400833	test	rax, rax
–	0x400836	je	0x40083d
–	0x400838	call	0x4009a0
–	0x40083d	add	rsp, 8
–	0x400841	ret	
_start	0x4009b0	xor	ebp, ebp
–	0x4009b2	mov	r9, rdx
–	0x4009b5	pop	rsi
–	0x4009b6	mov	rdx, rsp
–	0x4009b9	and	rsp, 0xfffffffffffffff0
–	0x4009bd	push	rax
–	0x4009be	push	rsp
–	0x4009bf	mov	r8, 0x402850
–	0x4009c6	mov	rcx, 0x4027e0
–	0x4009cd	mov	rdi, 0x402477
–	0x4009d4	call	0x4008f0
–	0x4009d9	hlt	
–	0x4009da	nop	word ptr [rax + rax]
deregister_tm_clones	0x4009e0	mov	eax, 0x6030d7
–	0x4009e5	push	rbp
–	0x4009e6	sub	rax, 0x6030d0
–	0x4009ec	cmp	rax, 0xe
–	0x4009f0	mov	rbp, rsp
–	0x4009f3	jbe	0x400a10
–	0x4009f5	mov	eax, 0
–	0x4009fa	test	rax, rax
–	0x4009fd	je	0x400a10
–	0x4009ff	pop	rbp
–	0x400a00	mov	edi, 0x6030d0
–	0x400a05	jmp	rax
–	0x400a10	pop	rbp
–	0x400a11	ret	
register_tm_clones	0x400a20	mov	esi, 0x6030d0
–	0x400a25	push	rbp
–	0x400a26	sub	rsi, 0x6030d0
–	0x400a2d	sar	rsi, 3
–	0x400a31	mov	rbp, rsp
–	0x400a34	mov	rax, rsi
–	0x400a37	shr	rax, 0x3f
–	0x400a3b	add	rsi, rax
–	0x400a3e	sar	rsi, 1
–	0x400a41	je	0x400a58

Figure 18 – GUI view

General Usage

- You may select functions to be dispplayed in the GUI by double clicking a function from the list on the left

- Clicking on a basic block in the user interface will show scroll to and highlight corresponding instruction in the instruction table to the right

- The instruction table can be searched for any of its components using the searchbar in the toolbar

- A CFG can be zoomed in and out of by pressing the zoom in or zoom out buttons on the toolbar

- You may export a CFG by opening the desired CFG, perhaps from the tabbed pane, and selecting ‘Export’ on the toolbar. This writes a file to your computer in the selected directory

- You may select any number of functions in the instruction table by clicking and dragging over the desired functions, or clicking the table and selected edit → select all

- If instructions in the table are selected, you may right-click them and select ‘export’ which brings up a new window with different export options
    - Export Function name, address, menemonic, and opstring
    - Export address, menemonic, and opstring
    - Export menemonic, and opstring
    - Export spaced hexadecimal representation of the instructions
    - Export non-spaced hexadecimal representation of the instructions.

- If instructions in the table are selected, you may alternatively right click and copy the instructions in one of the formats mentioned directly above to clipboard, but no for hexadecimal

Bibliography

I am still actively reading into the topic as a whole, devising a plan to implement an algorithm, along the lines of something like the disassembly algorithm used by BIRD.
(https://www.researchgate.net/profile/Lap_Lam/publication/4231624_BIRD_binary_interpretation_using_runtime_disassembly/links/00b49518db7561ed95000000/BIRD-binary-interpretation-using-runtime-disassembly.pdf)
This is the most fascinating proposal yet, but I chose not to include it in this report, because I still have a lot to understand before being able to go into depth!

Disassembly of Executables Revisited
Benjamin Schwarz, Saumya Debray, Gregory Andrews
University of Arizona Department of Computer Science, 2002
https://www2.cs.arizona.edu/~debray/Publications/disasm.pdf
Annotation:
In this publication, the Linear Sweep and Recursive Traversal techniques are discussed, with their positive and negativ aspects taken into account. A hybrid algorithm is suggested, based on the assumption that the executable contains a relocation table.

Obfuscation of Executable Code to Improve Resistance to Static Disassembly
Cullen Linn, Saumya Debray
University of Arizona Department of Computer Science, 2003
https://www2.cs.arizona.edu/solar/papers/CCS2003.pdf
Annotation:
This publication discusses the basic disassaembly algorithms, and obfuscation techniques. Elaborates nicely on how each algorithm can be neutered.

Linear Sweep vs Recursive Disassembling Algorithm
Dejan Lukan
Infosec Institute, 2013
http://resources.infosecinstitute.com/linear-sweep-vs-recursive-disassembling-algorithm/
Annotation:
This publication discusses how to break both Linear Sweep and recursive traversal, as well as outlining the main challenges of disassemblers.

Thoughts about disassembling algorithms
Dejan Ge0
Ge0-blog
http://ge0-it.blogspot.co.uk/2013/08/thoughts-about-disassembling-algorithms.html
Annotation:
A really nice and simple introductionary level blog on disassembly algorithms.

Understanding the ELF
James Fisher, 2015
Medium.com/@MrJamesFisher
https://medium.com/@MrJamesFisher/understanding-the-elf-4bd60daac571
Annotation:
Hands down the single article which ended any confusion I had around executable files. This was something I was stuck on for so long, honestly…I even tried to disassemble Mach-O executables from their 0th bytem thinking that was the way to do it…

What is the algorithm used in Recursive Traversive disassembly?

Answer by PSS, question by perror
Reverse Engineering Stack Exchange
https://reverseengineering.stackexchange.com/questions/2347/what-is-the-algorithm-used-in-recursive-traversal-disassembly
Annotation:

A question primarily outlining how IDA Pro conducts disassembly, and the different type of instructions that could be encountered.

ELF-64 Object File Formati

uClibc. 1998
https://www.uclibc.org/docs/elf-64-gen.pdf
Annotation:

The go-to document for reading in ELF files. Very informative, covering everything from table layouts in ELF files to data types.

Works Cited

https://blogs.oracle.com/ali/inside-elf-symbol-tables[8]
Benjamin Schwarz, S. D. G. A., n.d. *Disassembly of Executable Code Revisited ,* s.l.: University of Arizona. [4]
Cullen Linn, S. D., n.d. *Obfuscation of Executable Code to Improve Resistance to Static Disassembly,* s.l.: University of Arizona. [3]
Singh, A., n.d. *Identifying Malicious Code Through Reverse Engineering.* 2009th ed. s.l.:Springer. [6]
uclibc, May 27, 1998. *ELF-64 Object File Format.* [Online]
Available at: https://www.uclibc.org/docs/elf-64-gen.pdf [7]
Wikipedia, 2017. *Disassembler.* [Online]
Available at: https://en.wikipedia.org/wiki/Disassembler [1]

http://www.delorie.com/gnu/docs/binutils/as_265.html [2]

http://www.capstone-engine.org/BHUSA2014-capstone.pdf [5] Nguyen Anh Quynh, Coseinc, BlackHat USA: Capstone: Next-Gen-Disassembly Framework

http://security.di.unimi.it/sicurezza1112/slides/Lezione4.pdf [9] Alessandro Reina, Universita degli Studi di Milano ` Facolt`a di Scienze Matematiche, Fisiche e Naturali

https://www.usenix.org/legacy/publications/library/proceedings/sec04/tech/full_papers/kruegel/kruegel_html/disassemble.html [10] Christopher Kruegel, William Robertson, Fredrik Valeur and Giovanni Vigna Static Disassembly of Obfuscated Binaries https://www.omicsonline.org/open-access/should-reverse-engineering-remain-a-computer-science-cinderella-2165-7866.S5-e001.php?aid=12682 [11] Should Reverse Engineering Remain a Computer Science Cinderella? Christian Mancas

https://reverseengineering.stackexchange.com/questions/6455/what-are-the-targets-of-professional-reverse-software-engineering [12] What are the targets of professional reverse software engineering? – Answer by joxeankoret

https://www.gpo.gov/fdsys/pkg/PLAW-105publ304/pdf/PLAW-105publ304.pdf [13] Public US Law

https://en.wikibooks.org/wiki/Reverse_Engineering/Legal_Aspects#cite_note-1 [14] Reverse Engineering/Legal aspects

https://www.law.cornell.edu/uscode/text/17/107 [15] Limitations on exclusive rights: Fair use Cornell Law School

https://www.nytimes.com/2017/11/12/us/nsa-shadow-brokers.html [16] NSA secrets leaked. Scott Shane, Nicole Perlroth, and David E. Sanger

https://en.wikipedia.org/wiki/Atari_Games_Corp._v._Nintendo_of_America_Inc. [17] Atari V Nintendo Wikipedia

https://www.cs.ucsb.edu/~vigna/publications/2016_SP_angrSoK.pdf [18] The Art of War: Offensive Techniques in Binary Analysis