A Reverse Engineering Tool for Static Analysis Which Performs Equational Reasoning on X86 Assembly Code

Marien Bourguignon

Promoteur : Prof. Olivier Markowitch
Superviseur : Nikita Veshchikov

Mémoire présenté en vue de l’obtention du grade de Master en Sciences Informatiques
Acknowledgements

I would like to thanks Nikita Veshchikov, my supervisor, for the helpful advices he gave me throughout the long journey that is writing a thesis, but also for his many reviews which made my work what it is now.

I would also like to thanks Olivier Markowitch, my promoter and teacher, for letting me write on a topic of my choosing.

Finally, I want to give a very special thanks to Pabla Marmolejo, who, from the southernmost country on Earth, managed to give me the most valuable help in the form of love and hours of proofreading.
Abstract

Kevin Coogan and Saumya Debray, two researchers focused on digital reverse engineering, identified an issue within that field, and exposed it in a paper titled *Equational Reasoning on x86 Assembly Code*[1]. They stated that, while there is a great amount of tools able to perform reverse engineering analysis on high-level source code, there is a lack of such tool able to used on assembly code. The aim of this thesis is to show how the tool proposed in the aforementioned paper, which performs equational reasoning on x86 traces with the intend of improving their readability, could be extended to also perform static analysis. In this context, two additional issues have to be solved: Modelising the non-linear control flow, and deciding whether or not specific pointers are aliased. The former is solved using the static single assignment form, the later is handled thanks to a pointer analysis.

When performing manually what the static analysis tool would do, one can notice how the readability of its output has decreased compared to the one working on traces. This is due to the fact that the $\phi$-functions introduced by the static single assignment form does not clearly show which control structure has led to its existence, but also because of the undecidability of the pointer analysis problem, which implies that the used algorithm will only be able to provide approximative results.
## Contents

1 Introduction ................................. 1
   1.1 Introducing the Problem .................. 1
   1.2 Contribution ............................... 2
   1.3 Organisation ............................... 2

2 Digital Reverse Engineering ............ 3
   2.1 Introducing the Topic .................... 3
       2.1.1 A Piece of History .................. 3
       2.1.2 Definition ............................. 4
   2.2 Motivations and Scenarios ............... 4
       2.2.1 Software Engineering .................. 5
       2.2.2 Information Security ................... 6
       2.2.3 Scenarios ............................... 7
       2.2.4 Data Reverse Engineering ............. 8
   2.3 Legal Aspects .............................. 9
       2.3.1 Digital Millennium Copyright Act ........ 10
       2.3.2 Copyright Law and Fair Use .............. 10
       2.3.3 Trade Secret Law ........................ 11
       2.3.4 Contract Law ............................. 12
   2.4 Prerequisites .............................. 12
       2.4.1 Hardware ............................... 13
       2.4.2 Operating System ....................... 14
       2.4.3 Programming Languages ................. 16
           2.4.3.1 High vs Low Level Programming Languages .... 16
           2.4.3.2 Imperative vs Functional Programming Languages .... 17
       2.4.4 x86 Architecture ....................... 19
           2.4.4.1 Registers ........................... 20
           2.4.4.2 Instructions ......................... 21
           2.4.4.3 Syntax ............................. 22
       2.4.5 Compilers ............................... 22
       2.4.6 Executable File Format ................. 25
2.5 Performing Reverse Engineering

2.5.1 Level of Granularity

2.5.1.1 System Level Reversing

2.5.1.2 Code Level Reversing

2.5.2 Reversing Approaches

2.5.2.1 Offline Code Analysis

2.5.2.2 Live Code Analysis

2.5.3 Tools

2.5.3.1 Classification Tools

2.5.3.2 Disassemblers

2.5.3.3 Decompiler

2.5.3.4 Debuggers

2.5.3.5 Strings Detecting Tools

2.5.3.6 PE Analysis Tools

2.5.3.7 Tracing Tools

2.5.3.8 Monitoring Tools

2.5.3.9 Virtual Machines

2.5.3.10 Memory Scanning Tools

2.5.3.11 Hex Editors

2.5.3.12 Visualisation Tools

2.6 Obfuscation Techniques

2.6.1 Data-based Obfuscation

2.6.1.1 Constant Unfolding

2.6.1.2 Data-Encoding Schemes

2.6.1.3 Dead Code Insertion

2.6.1.4 Arithmetic Substitution via Identities

2.6.1.5 Pattern-Based Obfuscation

2.6.2 Control-based Obfuscation

2.6.2.1 Inline and Outline Expansion

2.6.2.2 Removing Sequential and Temporal Locality

2.6.2.3 Opaque Predicates

2.6.2.4 Interleaving Function’s Body

2.6.2.5 Processor Based Control Indirection

2.6.3 Combining Data and Control Flow Techniques

2.6.3.1 Junk Code Insertion

2.6.3.2 Control Flow Graph Flattening

2.6.3.3 Virtual Machines

2.6.4 Other Anti Reverse Obfuscation

2.6.4.1 Removing Symbolic Information

2.6.4.2 Anti Debugging

2.6.4.3 Confusing Disassemblers
3 Equational Reasoning on x86 Assembly

3.1 Algebra of Program ................................................. 63
3.2 Equational Reasoning of x86 Assembly Code .................... 64
  3.2.1 Motivating Example ............................................ 66
  3.2.2 Notation ....................................................... 67
  3.2.3 Implementation ................................................. 69
    3.2.3.1 Translating Instructions .............................. 69
    3.2.3.2 Resolving Dependencies .............................. 69
    3.2.3.3 Applying Equational Reasoning ..................... 70

4 Contribution .......................................................... 72
  4.1 Idea and Motivation ............................................. 72
  4.2 Complications .................................................... 72
    4.2.1 Branching ................................................. 72
    4.2.2 Indirect Memory Access .................................. 73
  4.3 Static Single Assignment ..................................... 74
    4.3.1 Control Flow Graph ...................................... 76
    4.3.2 Translating into SSA form .............................. 76
      4.3.2.1 Setting the $\phi$-functions ....................... 77
      4.3.2.2 Variable Renaming .................................. 79
    4.3.3 Memory Aliasing ........................................... 81
  4.4 Implementation and Difficulties ................................ 86
    4.4.1 Possible Implementation and Difficulties ............. 86
    4.4.2 Limitations ................................................ 90
      4.4.2.1 SSA Form ............................................ 90
      4.4.2.2 Indirect Memory Accesses ........................ 91

5 Conclusion ........................................................... 93
Chapter 1

Introduction

1.1 Introducing the Problem

Reverse engineering is a blooming methodology that owns its widespread acknowledgement to the analysis of hardware systems, a process which involves deciphering the design of finished engineered products, usually with the ambition of replicating them [39]. It can be observed in a variety of disparate domains such as in the traditional manufacturing industry, the semiconductor industry, the defence industry, the car industry, and so on.

The following definition of reverse engineering gives a good insight of what this practice means while being broad for it has been used by the American court in a variety of different cases.

**Definition 1.** *Reverse engineering is the process of extracting know-how or knowledge from a human-made artefact.*  
*The Law and Economics of Reverse Engineering [13]*

In this definition, two important points can be found. The first one is that reverse engineering is about rediscovering knowledge and nothing more. The second point is that the artefact under scrutiny has to be man made or in other words, it has to have been engineered by one or more people. In these conditions one can understand the added term reverse, implying that it is about going backward in the engineering process to extract information out of the artefact.

Since the advent of the digital era, the world has seen computer systems getting more and more ubiquitous. They can now be found in virtually every modern houses as well as in most companies where they usually take critical parts in ensuring their effective
functioning. Reverse engineering has then increased its scope to encompass the digital world, with more general end goals. Its use is mostly found in software engineering and computer security, not necessarily with the intent of making replications, but rather to gain a more detailed understanding of a specific system [40].

With digital reverse engineering gaining in popularity, methods have been developed to counteract this practice. These methods usually go in either of the two main directions: Confusing the tools used to perform the reverse engineering, or confusing the reverse engineers by means of obfuscation. Naturally, methods to counteract these anti reverse engineering methods have consequently arose. This gradual escalation has led to the creation of a myriad of tools aimed at both helping developers protecting their applications from reverse engineering by embedding anti reverse engineering methods, and helping reverse engineers to perform reverse engineering while circumventing potential protections.

1.2 Contribution

The contribution of this paper consists of a detailed explanation as to how one could extend the capabilities of an existing tool described by two researchers, Kevin Coogan and Saumya Debray, in a paper titled: Equational Reasoning on x86 Assembly Code [1]. Their tool provides a means to perform dynamic analysis of x86 assembly traces, with the purpose of countering obfuscation methods. The contribution explains how to allow the tool to operate in a static context.

1.3 Organisation

This work has been organised in five chapters. Chapter 1 introduces the topic of reverse engineering, it states the contribution of this work, and it lays down its organisation. Chapter 2 describes in detail the topic of digital reverse engineering, that is: What it is, why it is helpful, what are the legal aspects which have to be taken into consideration, what are the prerequisites that have to be mastered prior to performing reverse engineering, how to perform reverse engineering, and finally, how to counter reverse engineering. Chapter 3 describes a dynamic analysis tool proposed by Kevin Coogan and Saumya Debray, which serves as a starting point for the contribution. Chapter 4 contains the contribution. It deals with the static single assignment form, pointer analysis, and how one could potentially implement a tool based on the work presented in Chapter 3. Chapter 5 is the concluding chapter of this work.
Chapter 2

Digital Reverse Engineering

2.1 Introducing the Topic

2.1.1 A Piece of History

In 1989, a Japanese company specialised in the video game industry called Sega Enterprise released a gaming platform under the name of Sega Genesis. For a game to be released on the console, it had to either be developed by third party developers who had agreed on their licensing deal or by a subsidiary company of Sega. It was mandatory for the licensees to pay an extra $10 to $15 per cartridge\(^1\) over the hardware manufacturing costs and to make the licensor, Sega, their exclusive publisher [14]. As a result, it would have prevented third party developers from making games not designed for the Genesis. To enforce their business plans, Sega implemented a protection mechanism inside the console for it to reject unofficial cartridges, which was kept secret from the outside world.

During that era, another video game company named Accolade took the decision to port their PC games to the Genesis, but without agreeing upon the licensing deal. The main obstacle was for them to find a way to bypass the protection mechanism embedded in the console to allow their cartridges to be accepted as legitimate ones. Using specific analysis tools, the company successfully understood the inner working of the console and defeated the protection mechanism, which gave them the necessary knowledge to port their games without the authorization of Sega.

Thereafter, Sega sued Accolade for copyright infringement as the tools used to extract the knowledge out of the Genesis had to generate intermediate copies of what is contained inside the console’s memory. Accolade initially lost the lawsuit but appealed the verdict and the court, in the end, ruled these copies as fair use since they were not present in the final products, the cartridges.

\(^{1}\text{Cartridges are removable enclosures that contain video games.}\)
This story illustrates aptly the subject of this work as well as its implications. Should the reader be interested in a broader view of the story, he or she could investigate the following books: *The Ultimate History of Video Games* [14] and *Legal Battles that Shaped the Computer Industry* [16].

### 2.1.2 Definition

**Definition 2.** *Digital reverse engineering* is the process of extracting know-how or knowledge from a digital artefact.

The above definition of digital reverse engineering has been inspired by the one presented in the introduction. One can notice the disappearance of the human-made condition as computer systems are de facto human inventions, and the appearance of the word *digital*, which means that these artefacts are expressed by means of sequences of zeros and ones. A *digital artefact* can more concretely be anything that lives inside a computer’s memory, such as this work being in a pdf format, a network protocol, a program\(^2\), an executable file\(^3\), or even a process\(^4\).

### 2.2 Motivations and Scenarios

The use of reverse engineering in the digital world usually arises from two of the many areas of computer science: Information security and software engineering. They will both be described in this section, and then a list of generic scenarios that could non exclusively originate from both areas will be given.

Knowing where a certain project falls in can sometimes be useful as it helps the reverser looking for the appropriate set of reversing tools, whether or not seeking legal counselling would be appropriate, or even finding the right piece of literature that could best enlight his or her mind.

---

\(^2\)A program is a static sequence of instructions that represent a computation [45] or a executing computational process according to the context.

\(^3\)An executable (file) is a file that embodies the program in a way that makes it understandable for computers.

\(^4\)A process is a container for a set of resources used when executing the instance of the program (namely, the executable). It contains, amongst other things, the description of the computation to be performed [45].
2.2.1 Software Engineering

**Definition 3.** *Software Engineering* is

1. The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

2. The study of approaches as in 1.

*IEEE standard glossary of software engineering terminology* [41]

The above definition makes it clear that, at its roots, software engineering follows the same idea as engineering, with the difference that it is applied to the digital world. When performing software engineering, it is sometimes useful to go backward in the engineering process, that is, doing backward-engineering instead of forward-engineering. The definition below embodies this process, and three realistic scenarios are given to illustrate its uses.

**Definition 4.** *Software reverse engineering is the practice of analysing a software system, either in whole or in part, to extract design and implementation information.*

*Handbook of Information and Communication Security* [18]

Today’s technology is tomorrow’s legacy system. Even if a good documentation can lessen the need of reverse engineering, there is no magic solution to remove it from the equation. Indeed, developers and designers come and go, taking with them knowledge that is not necessarily explicitly written. It could also happen that some parts of a software application\(^5\) were outsourced/bought and the company responsible for making the parts does not exist anymore. Moreover, when software engineers are asked to extend an existing system, 50 to 90% of the time is spent on program understanding, which is not economically sustainable. It is thus easy to see why reverse engineering can be beneficial for the software engineering world [19, 18].

Development processes are widely used when dealing with software engineering. They help dividing the development of an application into segments that are put one after the other to form a waterfall from the waterfall model, a cycle as defined by the agile methodology, or any other composition. These segments usually go in the direction that

\(^5\)A (software) application is what is provided to the end-users, it is the product and its functionalities.
produces the desired object of the process, in part or in whole. It is called forward-engineering. One could, for whatever reason, want to go backward in that process, that is, from a segment; to try to extract information from the one that led to it using reverse engineering. Thus, in this situation, reverse engineering techniques could be used on an executable file, a source code, an UML diagram or any kind of result from segments that compose the used process [22].

An example worth noticing is the Millennium Bug, also called the Year 2000 Problem. In the early nineties, people started pondering about what would happen upon reaching the year 2000. Because dates were usually encoded using two digits, 2000 would have been indistinguishable from 1900, potentially creating bugs in computer related systems. This lead to the development of reverse engineering tools that could detect bugs from source files or executable [20]. Funnily enough, at the time of writing, a very similar situation called the Year 2038 Problem is being stressed upon the software industry. The 32bit signed date format in Unix can only go as far as 03:14:07 UTC on Tuesday, 19 January 2038. The very next second will make the date tip over to the 13 December 1901, possibly leading to yet another range of problems [21].

2.2.2 Information Security

**Definition 5.** The term information security means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide:

- integrity, which means guarding against improper information modification or destruction, and includes ensuring information nonrepudiation and authenticity;

- confidentiality, which means preserving authorized restrictions on access and disclosure, including means for protecting personal privacy and proprietary information; and

- availability, which means ensuring timely and reliable access to and use of information.


Reverse engineering applied with a information security point of view is broadly speaking about protecting information lying inside digital objects that have security implications. Hereunder will be given two scenarios which illustrate that specific use.
A fictional company whose name is irrelevant for this case has had its main server breached by a malware, making the company vulnerable to external cyber-attacks. To remedy the problem, the company tasks its most talented developers to find and eradicate the malevolent application as well as to understand how it made it so far, in order to prevent the scenario from repeating itself again. To carry out these tasks, they can look (amongst other things) at the logs of the operating system running on the server and the intermediate devices, but it might not bring complete answers to the questions. Applying reverse engineering to the malware is one of the solutions to find out precisely what has been done to the system, how to fully remove the malicious application and maybe how it made its way into the system. On the other side of the spectrum, a hacker working for a competing company might use reverse engineering to find security flaws in a specific operating system with the intent of using them to create malicious application able to bypass security measures.

As the state of the technology in software languages evolved, many developers moved to high level programming languages that offer strong abstractions over the underling system with the intend to be more productive. As a result, it is not uncommon to lose touch with what is actually happening under the hood. When dealing with certain tasks, it might be necessary to dive back in the dirty mud of the low level world to get a deeper understanding of the underling system, something very useful when performing reverse engineering from a security point of view. Indeed, what better place than in the foundation of the whole system one could find security flaws? The main idea behind using reverse engineering in computer security is that, to defeat a crook, it is necessary to think like one [19, 18].

2.2.3 Scenarios

Hereunder are listed nine general and straight forward scenarios that involve reverse engineering identified in Handbook of Information and Communication Security [18] and Reversing: Secrets of Reverse Engineering [17]:

- **Malicious software**: Reverse engineering is used both by developers of malicious software and developers of anti-virus software. On one side, they try to find flaws in systems using reverse engineering with the intend of using them to create malicious applications, and on the other side they use reverse engineering to understand the behaviour of the malicious applications and how to counter them.

- **Reversing cryptographic algorithms**: When the security of an algorithm entirely lies in its secrecy, successfully applying reverse engineering to it could be synonym of breaking it. Another situation arises when implementing known cryptographic algorithms. Because implementation details can have unexpected impact, it is advised to either check the source code or to reverse the executable to make sure everything is conformed to the specifications.
• **Digital rights management**: Internet being a new means of communication, media content providers had to digitise their products to follow the market. Digital information being very easy to move and duplicate led to the surge of piracy. As an attempt to counter this tendency, the media providers decided to add technologies in their products, which try to control the distribution of the content. They are called digital rights management (DRM) technologies. Once again, crackers (reversers specialised in breaking digital protections) appeared on the opposite side of the spectrum.

• **Auditing executable files**: When the source code of an application is missing, all there is left are the executables. If one wants to audit these files to find vulnerabilities, reverser engineering is the only way to go.

• **Evaluating software quality and robustness**: When an end-product, which does not provide its source code, has to be audited to check if it qualifies to a certain level of quality and robustness, reverse engineering tools usually come to play.

• **Legacy software maintenance, re-engineering, and evolution**: Recover the design of a legacy system, usually when source code is no longer available, to allow maintenance, re-engineering, and evolution of that system.

• **Verification that implementation matches design**: As explained above, going backward in the development process of an application can give insight on the previous segments, which is useful to determine if the implementation matches the design.

• **Achieving interoperability with proprietary software**: Because, no matter how hard one tries, documentation is almost always insufficient when dealing with closed source system. It is often necessary to contact the vendor to get answers, but reverse engineering could be used to get answers faster.

• **Developing competing software**: One could try to steal an un-patented secret algorithm from a competitor’s product using reverse engineering. To be noted that, most of the time, software applications are too big to be entirely reversed, making it easier to start a new application from scratch.

2.2.4 **Data Reverse Engineering**

As mentioned previously, the definition of reverse engineering is vague on the kind of object that can be under observation. This work will be mostly about reversing techniques applied to executable files, and it will sometimes imply the use of data reverse engineering, which consists in deciphering program data such as the structure of a database, a network protocol, a data structure, or even a file format. To be more
precise, this work will be also interested in deciphering data that one can find inside an executable file.

An example of data reverse engineering can be found in the history of the .DWG proprietary file format developed by Autodesk [44]. At one point in time, it was mandatory to buy their software to use files encoded in that specific format and so, an association of software developers and users decided to reverse engineer it to create an open source alternative to work on that format [24]. The same things happened to file formats produced by the Microsoft Office suite.

Data reverse engineering can be useful both when performing reverse engineering with motivations from the information security or the software engineering field. The two following scenarios give insight on the matter:

- **Software engineering**: When reversing applications that are built to use databases or any kind of structured file, one could try to reverse engineer these objects to get insight on how the application works [23]. Because there is no logic embedded inside, it can be easier than coping with the whole system.

- **Computer security**: Understanding how data is represented inside an application’s memory can be useful to speed up the reversing process. An example would be understanding how an important data structure for the process to carry out its operations is structured, which would give insight on how the code manipulating it would look like.

### 2.3 Legal Aspects

Whether performing reverse engineering on a digital object is legal or not cannot always be easily answered as regulations differ on a country basis and do not even always have a straightforward answer. Laws such as the fair use in the United States are subject to interpretation and so, can only be answered by the court on a case by case basis. The introductory case of Sega vs Accolade is a perfect example of such situation, where the intermediate copies have been ruled as fair use, but only after appealing the initial judgement.

Extraction of knowledge from an artefact can be costly or cheap and time-consuming or fast [13]. The artefact and these notions are what determine if additional legal protections are necessary. The goal here is not to give an exhaustive list of all of these legal protections because it is not the purpose of this work, but rather to mention some of the main legal doctrines that could prevent a reverser to do his or her job in compliance with the law. The emphasis will be put onto the United States as information on the
matter is significantly harder to find for a country such as Belgium. Nevertheless, it can give a general idea of what could get in the way of a reverser.

2.3.1 Digital Millennium Copyright Act

“To pass laws that regulate the research of technological measures that protect copyrights and the dissemination of such results is to concede that copyright technology is broken and can never be improved — that the only possible outcome of allowing common people to understand copyright control technology is the demise of the technology.”

Andrew Huang, 2003

For years, copyright industries would sell their products in the form of tangible goods such as books and CDs. The rise of digital technologies opened up a new market for these industries with the possibility to the mass-marketing of contents that are technologically protected. At the same time, these companies pushed the American Congress to implement legal obstacles to protect the technological protections so that it would be illegal to break them. The Digital Millennium Copyright Act, or DMCA for short, is the law which embodies that legal protection [13].

According to the Electronic Frontier Foundation, this law does not only prevent breaking protections, but also breaking access controls. They give as example breaking authentication handshakes, code signing, code obfuscation, and protocol encryption [27]. The law has nevertheless an exception which allows the development and the use of tools to bypass these protections as long as it stays in the scope of interoperability [13].

2.3.2 Copyright Law and Fair Use

Copyright laws give a certain set of exclusive rights for an original work to its creator, the copyright owner. It is some sort of intellectual property that is applied to, but not only, software application. To make copies of a protected product, it is necessary to either have an agreement with the owner or to go through an exception granted by the copyright laws. Copyright does not prevent someone else from reinventing the protected object.

Most of the software applications are distributed in the form of digital objects because users don’t necessarily care about their source code representations and also because companies behind software want to keep their source code and associated documentation as trade secrets [26]. Decompilation and disassembly, two major techniques to perform reverse engineering that will be discussed later, could arguably infringe copyright laws as they make intermediate approximate copies of the original source code.
Fair use is one of the exceptions that can be used to make copying a software lawful. It allows a rightful owner of a copy of the software application to copy the work for a purpose and to an extend that will not hurt the owner of the copyright. The following list of factors are used to determine whether a certain application can fall into fair use [26]:

- the defendant’s purpose in using the protected work
- the nature of the copyrighted work
- the amount and substantiality of what is taken
- the potential for harm to the market for the protected work

Two more privileges are given to the owners of copies of copyrighted software. An essential step in launching a software is to copy the digital object into the random access memory (RAM) and then the caches. It is stated that if the copy has been lawfully acquired, that form of reproduction is not unlawful under copyright ground. Backup copies are treated the same way and under the same prerequisite. Copying a software application to reverse engineer it is going beyond these two privileges and might infringed the law if it does not fall into fair use or any other relevant exception [15].

2.3.3 Trade Secret Law

A trade secret is defined by the Uniform Trade Secrets Act as follows:

**Definition 6.** *Trade secret means information, including a formula, pattern, compilation, program, device, method, technique, or process,*

- *that derives independent economic value, actual or potential, from not being generally known to or readily ascertainable through appropriate means by other persons who might obtain economic value from its disclosure or use; and*

- *is the subject of efforts that are reasonable under the circumstances to maintain its secrecy.*

*CIVIL CODE SECTION 3426-3426.11 [43]*

Trade secret laws only protect from wrongful acquisitions and use or disclosure of trade secrets. An example would be breaching a non disclosure agreement or using industrial espionage. If the intermediate copies are ruled as fair use, obtaining the trade secrets using reverse engineering is in accordance with the law. The gathered information can thereafter be published and or used freely in the eyes of the law [15, 13, 16].
Since trade secrets are not perceived as intellectual properties (i.e. a monopoly over something to an owner designated by the law), one might want to consider patenting a discovery. To get a patent, an author has to disclose information containing at least a written description of the discovery and a series of steps to reproduce it. If the patent is accepted, it will fall into the public domain and so render the usage of reverse engineering useless as the knowledge will be freely available for all. In contrast with copyright, patents do prevent artefacts to be reinvented by someone else, but these protections have expiration dates.

2.3.4 Contract Law

Software can be sorted into two categories, free and not free software. The free has to be interpreted as freedom of speech and not as in free of charge. According to the Free Software Foundation, for a software to be free, it has to give a user the freedom to run, copy, distribute, study, change and improve itself [28]. On the other side, proprietary software strip users from certain liberties that are specified in end-user license agreements, or EULAs for short. When an application comes with an EULA, the reverser has to make sure it does not have a no reverse engineering clause as it legally prevents it. To be noted that the enforceability of these restrictions have been challenged in America as well as in Europe [13].

The same logic can be applied to services provided with a Terms of service, or TOS for short, or any other kind of contracts bundled with software that has to be agreed upon before use.

Figure 2.1: Part of the EULA bundled with the strings application provided by Microsoft.

2.4 Prerequisites

Before diving into the world of reverse engineering, it is important to be knowledgeable about a few prerequisites. This section will be focused on giving broad explanations on these topics as they could arguably be considered as the knowledge baseline over which someone can start performing reverse engineering. A deep understanding is not required, but, as one can expect, more knowledge comes with more ease, meaning work could be done more efficiently and with less headaches.
2.4.1 Hardware

The hardware is the foundation over which the software world lies. Digital objects are made of zeros and ones for computers live in a binary word, and therefore are unable to grasp anything beyond these two states. Understanding how a computer is made, and also how they use these successions of binary values to perform computations is important as reverse engineers usually have to work at or near this level of abstraction.

A very simple and outdated but still relevant architecture of a computer is the Von Neumann architecture. It was invented in 1945 by a Hungarian scientist named John von Neumann, and contains everything that is expected to be found in a mainstream computer: A CPU, a memory, and an input/output mechanism. A schematic representation can be observed in Figure 2.2.

![Von Neumann architecture](image)

The Von Neumann architecture works as follows: The CPU (Central Processing Unit) is fed instructions from the memory (MEM on the diagram) by the common bus (the arrows), and it performs the specific actions according to the opcodes and the operands found in the instructions. An opcode is a number that represents a function the CPU can perform, the operands being the possible parameters to be applied to that function. See Table 2.1 for an example. The execution of an instruction is characterized by either modifying a piece of data, displacing data from one component to another or enabling a specific functionality in a component. The common bus allows the three components to communicate with each other, but the CPU is the one making the calls in a way that follows the semantics of the instructions that are fed to it. With this model, data and instructions are stored in the same memory. This is still the case today as executable files contain parts of their data along the side of the instructions that compose them.

The Instruction Set Architecture, or ISA for short, is what is provided by the CPU to the software applications. It includes, amongst other things, the list of opcodes.
(instructions), how native data types are defined, the names of the registers along with their sizes and types, the addressing modes and the memory architecture, and how interruptions and exception handling are done. An ISA defines what a CPU can do as it is the only interface to the hardware that is given to the software applications [29]. As a result, it is part of what a low level reverser has to master. To be noted that the ISA can differ from one CPU to another.

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>A8 02</td>
<td>1010 1000 0000 0010</td>
<td>Compare the value of register al with 2</td>
</tr>
<tr>
<td>89 CB</td>
<td>1000 1001 1100 1011</td>
<td>Move the value of register ecx to ebx</td>
</tr>
<tr>
<td>83 F0 09</td>
<td>1000 0011 1111 0000 0000 1001</td>
<td>Xor the value of register eax with 9</td>
</tr>
</tbody>
</table>

Table 2.1: Intel x86 instructions made of an opcode (red) and two operands (blue).

An improved model called the Harvard architecture can be seen in Figure 2.3. It provides a separation between the data and the code by means of two different memory blocks and by doing so, it removes the Von Neumann bottleneck of the single shared bus of the first model. Nowadays, a third model called the Modified Harvard Architecture took over by combining the advantages of the two others: Instructions are treated as data while allowing concurrent instruction/data access. It is implemented as a hierarchy of caches that can be accessed concurrently over a monolithic memory that contains object code where code and data are mixed.

2.4.2 Operating System

A computer system can be roughly divided into four parts [30]: The hardware, the operating system, the software applications, and the users. An abstract representation of such system can be observed in Figure 2.4. As seen previously, the hardware is composed of at least a CPU, a (hierarchy of) memory, and an input/output mechanism that are used to carry out software applications. The software applications are tools used by the users to solve problems using the resources given by the hardware. Finally, the operating system is what controls the hardware and coordinates its use amongst the software applications, it does not produce useful work, but rather provides an environment that is used by software applications to do useful work. An operating system can be observed with two different viewpoints, from the user’s and from the computer’s viewpoint. They will both be briefly discussed hereunder.

From the user’s point of view, what matters in an operating system is how easier it makes the computer to use as well as the performances he or she can get from it,

See John Backus’ award winning lecture: Can programming be liberated from the Von Neumann style?: a functional style and its algebra of programs [63]
without caring too much about resources utilisation. When designing an operating system, it is important to take into account these two viewpoints to come up with the most appropriate compromises according to the purpose of the operating system being developed.

From the computer’s point of view, the operating system is supposed to be a fair and efficient resources allocator. It has to manage the resources given by the hardware such as devices, memory space, and CPU time by deciding how to allocate them to satisfy the needs of the software applications and the users. According to what the computer system is designed for, the definition of fair and efficient will vary.

As explained above, the operating system provides an environment for the software applications to do useful work. It gives an abstraction layer that hinders applications from having to take into consideration what kind of hardware is making a specific computer outside the CPU. Because the ISA is the only connection between the hardware and the software, the operating system cannot offer any abstraction on that regard. Some operating system also offer additional services such as inter-process communication, file management, computer administration, and so on. The communication between the
applications and the operating system is mainly done through system calls. Understanding how these calls are made as well as their repercussion on the system is of critical importance from a reverser engineering’s point of view.

2.4.3 Programming Languages

**Definition 7. Programming languages are notations for describing computations to people and to machines.**

*Compilers: Principles, Techniques, and Tools [32]*

Computers and human beings do not speak the same languages, the former can only comprehend numbers whereas the later feel more comfortable around words and sentences. Programming languages are the mean used to instruct a computer on what to do using formally constructed syntaxes that are understandable by humans. The key point here is that these languages follow strict rules that allow automatic translation from one representation to the other. A source code is obtained by writing instructions using a programming language. Most of the time, these pieces of texts are processed by automatic translation tools called compilers to generate a semantically equivalent but syntactically different sequence of instructions. See Section 2.4.5 for more information on compilers.

As the ISA is the bridge between the software applications and the hardware world, programming languages can be seen as the bridge between human beings and the hardware world. These languages come in many forms, ranging from high level to low level, from imperative to purely functional, and many other classifications. In this work, the two previously cited classification are of great importance as they will be mentioned in the subsequent pages.

2.4.3.1 High vs Low Level Programming Languages

The more a programming language abstracts away the ISA and the system calls of the operating system, the more high level it is. Assembly languages are considered to be the lowest level programming languages there are as they only translate opcodes that are expressed with numbers to Latin characters. On the opposite side of the spectrum, programming languages such as functional languages are considered to be high level because the underlying mechanism of the computer does not appear in the source code. For productivity reasons, one would want to use a high-level programming language because he or she can do more operation by writing less code. For efficiency reasons, one would rather want to use a low-level programming language as it allows optimisation
on a finer grain scale. There is an abundant variety of programming languages that vary in position on this spectrum, one has to choose wisely according to its needs.

In Listing 2.1 and Listing 2.2 can be observed two source codes, one written in an assembly language called Microsoft Macro Assembler, or MASM for short, and the other in Haskell. They both perform the same operation, displaying the sequence of characters “Hello World!” on the screen, but they differ in the syntax. Haskell, being more high level than MASM, allows to express instructions in a more human understandable way whereas in MASM, it is barely understandable.

```
.MODEL Small
.STACK 100h
.DATA
  db msg 'Hello World!$
.CODE
  start:
    mov ah, 09h
    lea dx, msg
    int 21h
    mov ax, 4C00h
    int 21h
  end start
```

Listing 2.1: Hello World written in MASM

```
main :: IO ()
main = do putStrLn "Hello World!"
```

Listing 2.2: Hello World written in Haskell

2.4.3.2 Imperative vs Functional Programming Languages

Human languages such as English and French are used to communicate with one another. They provide grammatical moods that are most of the time used in conjunction with verbs to express the attitude of a speaker toward what he or she is saying. One of such moods is the imperative which expresses commands or requests. For example, in the sentence “Write your thesis”, the speaker use the imperative to order him or herself to get back to work.

Similarly, imperative programming languages use statements which change the program’s state. An example of source code written using an imperative language named C can be observed in Listing 2.3. It declares a function called `fib` which takes an integer `n` and returns the `n`th Fibonacci number. Inside its body can be observed a sequence of imperative statements which tell the computer how to change its state. The structured constructs of selection (if) and repetition (for) allow the programmer to direct the flow
of execution at run time according to previous states.

```c
int fib(int n) {
    int p_fib = 0;
    int c_fib = 1;
    int tmp, i;

    if (n == 0)
        return p_fib;

    if (n == 1)
        return c_fib;

    for (i=1; i<n; i++) {
        tmp = c_fib;
        c_fib = p_fib +
                c_fib;
        p_fib = tmp;
    }

    return c_fib;
}
```

Listing 2.3: Fibonacci written in C

```
— Inefficient implementation
fib :: (Num a, Eq a) => a -> a
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n -2)

— Efficient implementation
fib :: Int -> Integer
fib n = fibs !! n
fibs :: Num a => [a]
fibs = 0 : 1 :
      zipWith (+) fibs (tail fibs)
```

Listing 2.4: Fibonacci written in Haskell

Functional programming languages offer another approach to direct a computer. They
describe to the computer what is desired instead of how to do it. As it can be observed
in Listing 2.4, instead of giving the steps the computer has to follow to generate the
nth Fibonacci number, it tells that a Fibonacci number is either 0, 1, or the sum of
the two direct precedent Fibonacci numbers. Functional languages differ fundamentally
from the imperative ones as they do not allow side-effects by preventing states from
changing. A variable, when initialised to a specific value, cannot see its state (i.e value)
changed. This can be generalised to function as they will always produce the same
result when applied to the same parameters. More formally, \( f(x) = f(x) \) is always true.
An interesting implication is that programmers don’t have to think about control flow
anymore as the order of execution becomes irrelevant\(^7\). These languages are said to be
referentially transparent, a property that allows equational reasoning. See Section 3.1
for more information on equational reasoning and see the influential paper entitled Why

\(^7\)This does not tackle the problem of data dependency.
Functional Programming Matters [33] written by John Hughes for more information on functional programming.

2.4.4 x86 Architecture

x86 is a family of ISA that is backward compatible with many of Intel’s processors. It was first released in 1978 inside the 8086 processor and then continued to be used and extended in the subsequent processors such as the 80186, 80286, 80486 and many others. The name x86 comes from the fact that for a period of time, most of Intel’s processors which had an architecture belonging to that family had names finishing in 86, thus x86.

Since x86 is backward compatible, it encompasses the 16-bit, 32-bit and 64-bit version of the architecture. The 64-bit architecture is called x64, AMD64 for it was first introduce by AMD, or even x86_64 because it is an extension to the x86 architecture. It was released in 2000 [35]. The 32-bit one is also called IA-32 for "Intel Architecture 32-bit" and was released in 1985. The IA-64 does not correspond to x64, it actually refers to the 64-bit Itanium architecture of Intel.

The $x$ in “$x$-bit architecture” roughly means how many bytes are addressable, or in other words, what is the maximal size of the address space (which is the range of memory that can be addressed). The addresses, when used by the CPU, have to be stored in registers. They then have to have a size at least equal to that $x$. For example, a 32-bit architecture can address $2^{32}$ bytes and has to have 32-bit wide registers to store these addresses.

This family of ISA is said to be little endian. Endianness refers to the order in which the bytes that compose a in-memory multi byte value are ordered. With big endian, the most significant byte is stored first at the lowest address. Little endian is the opposite, the most significant byte is stored at the highest address. See Table 2.2 for two examples.

<table>
<thead>
<tr>
<th>Value</th>
<th>Big-endian</th>
<th>Little-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1CEB00DA</td>
<td>1C EB 00 DA</td>
<td>DA 00 EB 1C</td>
</tr>
<tr>
<td>0xDEADC0DE</td>
<td>DE AD C0 DE</td>
<td>DE C0 AD DE</td>
</tr>
</tbody>
</table>

Table 2.2: Differences between little and big endian.

The IA-32 can operate in two modes, real and protected. In real mode, the processor has to be used as if it only supported 16-bit instructions. In that mode, the processor allows unrestricted memory accesses to all the running processes. The protected mode does not have the 16-bit restriction and provides virtual memory, paging, and subsequently protection over memory locations. Nowadays, CPUs start in real mode
for compatibility reasons and are switched to protected mode by the operating system after having done a specific initialisation.

x86 also provides levels of privileges ranging from 0 to 3 called ring levels [36]. Ring 0 is the highest privilege level, giving unrestricted access to the system, ring 1 and 2 are usually not used, and finally ring 3 allows restricted read and modification of system settings. Today’s operating systems usually implement privilege separations by means of these rings, ring 0 being kernel mode (for the operating system) and ring 3 user mode (for the user’s applications).

The following sections will discuss the x86 ISA in user mode in its 32-bit version. It is important to master this topic as this document will solely focus on this family of architectures.

2.4.4.1 Registers

Registers are units of memory of bounded size that are used by the Arithmetic and Logic Unit, or ALU for short, to store operands and results of instructions. They have specific names to allow discrimination from one another as they are not referenceable using memory locations. For speed matters, they are located very near the ALU, and are built in such a way that loading and storing values takes as little clock cycles as possible.

IA-32 has four 32-bit general purpose registers being eax, ebx, ecx and edx; five index and pointer registers being esi, edi, ebp, eip and esp; and one flag register eflags. x86 being backward compatible, it is still possible to use the 16-bit versions of these registers by omitting the leading e that stands for extended. They do not refer to another register but rather the first 16-bit of the corresponding extended register. The four general purpose registers can be furthermore subdivided in their non extended form by replacing the x by either h or l. They respectively stand for high for the most significant 8-bit and low for the least significant 8-bit.

Intel’s engineers gave names to the registers according to their purposes. Some are still used the way they were designed to be, others not as much as before. The most flagrant example would be the four general purposes registers eax, ebx, ecx and edx, all of which are optimised to be used in specific situations but can very well be used interchangeably. Still, using the right register with its corresponding instruction is interesting for compression and documentation reasons. Some instructions are built in such a way that they implicitly use specific registers, making it useless to specify them when the instruction is called. Once one knows about these purposes, he or she can also get a faster understanding of the program as the code will be more or less self documenting [27]. The registers and their names will be discussed hereunder.
- **eax, ax, ah, al** is the Accumulator register, hence the *a*. It is optimised to be used as an accumulator.

- **ebx, bx, bh, bl** is the Base register, hence the *b*. It used to be one of the few registers that could be used as a pointer. It lost its function as most registers can be used this way.

- **ecx, cx, ch, cl** is the Counter register, hence the *c*. Again, optimised to be used as a counter.

- **edx, dx, dh, dl** is the Data register, hence the *d*. It is an extension to the accumulator.

- **esp, sp** contains the address of the top of the current stack. Its name comes from (extended) stack pointer.

- **ebp, bp** contains the base address of the current stack. Its name comes from (extended) base pointer.

- **esi, si** contains the source address for string and memory operations. Its name comes from (extended) source index.

- **edi, di** contains the destination address for string and memory operations. Its name comes from (extended) destination index.

- **eip, ip** contains the address of the next instruction to execute. Its name comes from (extended) instruction pointer.

- **eflags, flags** contains the state of the processor by means of binary flags. Most instructions have implicit effects on these flags, and they can be used to do conditional branching. See Intel’s manual for a complete description of that register.

There are also five segment registers used to do segmented addressing. They are called **cs** for code segment, **ds** for data segment, **ss** for stack segment, and finally **es**, **fs** and **gs** that are extra segments at the disposition of the programmers. Segmentation allows a program to be split into segments that have independent address spaces [34]. A usual way of doing this is to have the code (i.e the sequence of instructions) and the stack separated, each of them in their own segment.

### 2.4.4.2 Instructions

The instructions provided by the ISA have two forms, the one understood by the CPU and a symbolic one understood by humans. The later has the following format [34]:

[label: mnemonic argument1, argument2, argument3]

Label is an identifier followed by a colon, the mnemonic is a reserved name for a class of instruction opcodes with the same function, and the arguments (or operands) are what
is applied to the function. A function can have from zero to three operands, which can be either literals or identifiers for data items. When a instruction is taking only two operands, the right one is the source and the left one is the destination.

2.4.4.3 Syntax

There are two syntax notations for assembly code written using the x86 instruction set, the Intel and the AT&T syntax. As it is suggested, they only differ in the way of representing the same thing. The biggest differences are the following:

- AT&T prefixes registers with the symbol % and immediate values with $. On the contrary, Intel does not use anything to differentiate the two.
- Intel puts the destination operand on the left, whereas AT&T puts it on the right.
- Intel does not use different mnemonics for the same instruction applied to operands of different size, while AT&T does.

An example of the same assembly code written with the two syntaxes can be observed in Listing 2.5 and Listing 2.6. This work will solely use the Intel syntax.

![Listing 2.5](image1.png)

Listing 2.5: Equivalent of Listing 2.6 using the Intel syntax.

![Listing 2.6](image2.png)

Listing 2.6: Equivalent of Listing 2.5 using the AT&T syntax.

2.4.5 Compilers

**Definition 8.** A **compiler** is a program that can read a program in one language — the source language — and translate it into an equivalent program in another language — the target language.

*Compilers: Principles, Techniques, and Tools [32]*

A compiler could be compared as a human translator doing textual translation from a source natural language to a target natural language. The translator has to choose its words wisely as it is important for the resulting translation to have the same meaning as the original text. To do so, a translator has to understand the meaning of the source

---

*An American multinational telecommunications corporation.*
text as well as its context and to stay as faithful as what the original author wrote. Compared to human translators, compilers are not always able to translate source texts as faithfully as a translator would because of ambiguities arising from the contexts. Programming languages are languages that have formal sets of rules to unambiguously define what makes a well-formed source code called grammars. Compilers use (amongst other things) these grammars to generate semantically equivalent translations in a target language.

Compilers can technically translate from any programming language to another, but they are more frequently used to translate from one specific language to an assembly language. As it has been explained in Section 2.4.3.1, most programmers will prefer to work with programming languages that offer layers of abstraction over the underlying system. The purpose of compilers is therefore to automatically remove these layers, or in other words, to specialise a source code for the hardware and the operating system to understand it.

The compiling process is the process in which a source code is turned into an executable file. It is made of multiple tools (from which the compiler belongs) that are put one after the other to gradually perform the transformation. The process can be observed in Figure 2.5. To turn a source code written using a specific programming language into target machine code (i.e. turned into understandable machine instructions in a structured file for the computer to execute), the subsequent steps are usually followed:

1. **Preprocessor**: The preprocessor can be used to, amongst other things, gather all the source code files making the application and to merge them into one file, to do textual swaps, to do macro expansions, and to extend the underlying language.

2. **Compiler**: It does the translation from one language to the other. In this situation, from a specific language to an assembly language. The output will be made while taking into account the ISA as well as the operating system of the targeted system. Compiling can be divided into two phases, the analysis and the synthesis. The analysis breaks up the code of a source file into tokens and tries to find out the structure of the code using the grammar. If it succeeds, it will then check if the structure makes sense semantically. If yes, the analysis part is over and the result is sent to the synthesis. However, if the code is syntactically incorrect or semantically unsound, the process can’t continue. Upon receiving the output of the analysis, the synthesis part will generate the code for a targeted platform in the form of assembly code.

3. **Assembler**: It will textually translate assembler instructions into the opcode/-operands dyads by doing lookups on the mnemonics. The resulting object is called an object file. It is important to understand that machine instructions have a one-to-one relationship with assembly instructions.
4. **Linker**: It will merge all the object files that made an executable file while resolving addresses pointing from one object file to another.

Figure 2.5: A language-processing system. Image inspired by *Compilers: Principles, Techniques, and Tools* [32].

From a reversing point of view, understanding the compilation process and even more so the compiler is of great importance. The translation into an executable file is not a direct translation, one operation can be mapped to a gaggle of other operations, and code optimisation can be carried out, meaning that some parts of the original code can be modified, reordered or even deleted. Moreover, the process is lossy, meaning that variable and function names are usually lost and that variable types are no more clearly identifiable, and it is a many-to-many operation because a source program can be translated into machine code in many different ways, and also the other way around [10]. The insight one can get by understanding this process, and all that comes with it, can be very useful when performing reverse engineering.

Some programming languages do not go all the way down to the compilation process until runtime. This family of languages includes Java or C# which use an intermediate representation called bytecode or MSIL. These intermediate languages usually keep
plenty of information that would not be found if the compilation was done once and for all.

2.4.6 Executable File Format

An executable file format is a standardised way of representing an executable file so that the loader can process it. In modern operating systems, an executable file cannot be simply plastered into RAM memory for the CPU to execute it. The loader will first have to parse the structure to extract meta information about the executable to set up and manage the adequate environment in which the program found in the executable will be executed.

There are many structures as they are usually different across operating systems. Windows is using the Portable Executable (or PE) file format which is an extended version of the Common Object File Format, or COFF for short, developed by AT&T since Windows NT 3.1 [37]. Many Unix-like operating systems use the Executable and Linkable Format, or ELF for short, developed by the Unix System Laboratories [46].

![Figure 2.6: Simplified representation of the ELF format. Image inspired by the Tool Interface Standards [46].](image)

In Figure 2.6, one can observe how files formatted using the ELF structure are organised. The segments, as explained above, contain what concretely makes the executable, the rest is metadata. The ELF header describes the file’s organisation, the program header table tells the system how to create a process image, and finally the section header table contains information about the segment.

In the metadata, one can usually find the entry point of the executable (the place the processor has to start decoding and executing instructions), where the different
segments are located, the time at which the linker produced the file, the type of the file (executable, dynamic-link library, ...), and so on.

2.5 Performing Reverse Engineering

2.5.1 Level of Granularity

There are many different approaches to start reversing. They can be sorted into two categories, system-level reversing and code-level reversing. They differ according to the level of granularity provided by the analysis, the finest grained ones being not only more complete, but also more complex to realise and understand.

2.5.1.1 System Level Reversing

System level reversing is focused on extracting information from a software application through its side effects on the system. As the operating system is the layer of abstraction that prevents software application from having to bother with the dirty details of the underlying hardware, everything has to go through it. Monitoring the effect of a program at a system level can provide a lot of information without the requirement of diving into a pool of assembly lines.

2.5.1.2 Code Level Reversing

Code level reversing, as its name suggests, consists of looking at the code (i.e the machine instructions, the Java code, ...) to extract the needed information. Since the code is what instructs the system on what to perform, code level reversing is more general than system level reversing, but the major drawback is that it is significantly harder to perform the former. Offline and live code analysis, two approaches discussed in 2.5.2, fit in this category.

2.5.2 Reversing Approaches

The process of reversing an executable file can be tackled in many different ways. Depending on what the reverser is looking for, its knowledge, the tools it has in its disposition, and also the legal aspects that could be surrounding the software application under examination, he or she has to find the right approach(es) that suit(s) best that specific need. They will be discussed hereunder.

2.5.2.1 Offline Code Analysis

As said before, machine instructions have a one-to-one relationship with assembly instructions, and as a result they can be translated back and forth relatively easily. Offline code analysis consists of translating the machine instructions of an executable file into assembly and inspecting the resulting readable assembly code. The translation from
machine instructions to assembly instructions is called disassembly, and it is performed by a disassembler.

The main interest of offline code analysis is that the code does not have to be run to be analysed. The reverser can read the generated code to try to find parts that are relevant to his or her analysis. One drawback is that it does not show the control flow of the program nor the data that is being manipulated as the program advances in the instructions, which makes this approach significantly harder than the others. It happens that, when an executable file is protected with the right technological protection, offline code analysis is not possible. The reason is that the machine instructions could be obfuscated in such a way that they only appear in their right forms during runtime (or for a certain amount of time during runtime). This will be later explained in Section 2.6.

2.5.2.2 Live Code Analysis

This approach also uses the output of a disassembler, but with the addition that it thereafter runs it on what is called a debugger. These tools are used to see how the program evolves as it goes through the instructions, showing the control flow and how the data is being manipulated. This approach is considered easier to perform compared to the offline analysis, but requires the executable file to be run. According to the kind of executable being reversed, a virtual environment could be set up to prevent any damages on the system. More information on virtual environments can be found in Section 2.5.3.9.

2.5.3 Tools

2.5.3.1 Classification Tools

The first thing a reverser has to do when confronted to a binary file is to find out its nature. In the family of Windows's operating systems, most files have extensions that help answering that question but are by no means one hundred percent accurate. An example would be an executable file that ends with the “.exe” extension. Replacing it by one of an image, “.png” for example, won’t magically make a picture out of the executable. The example could be extended for any other types of files. A rule of thumb is to never fully trust extensions and to use classification tools that give more than educated guesses. Still, it is wise to not fully rely on the output of a tool, and to corroborate findings amongst many of them if possible. Two of such tools will be briefly presented directly below.

The file command can be found in Cygwin\(^9\) as well as many other UNIX/Linux operating systems. Given a sequence of one or more files, it will try to classify them. To

\(^9\)A Linux-like environment for Windows. See [https://www.cygwin.com/](https://www.cygwin.com/).
do so, it performs three series of tests: file system tests, magic tests, and finally language tests. The command will stop testing as soon as one test yields a match. The file system tests consist of asking the underlying operating system if the file has a special use for it or if it is empty. The magic tests check for numerical or textual values that are unique to specific file types (an example would be the compiled Java classes that are known for having 0xCAFEBABE as their first four bytes). Finally, if the two previous series of tests fail, the command will look for human readable text by checking usual character encodings. If nothing can be said, the command will simply say it contains data. An example of use can be found in Figure 2.7.

```
Marien@DESKTOP-NND4UGJ ~> $ file chile
chile: PNG image data, 1920 x 1079, 8-bit/color RGB, non-interlaced
Marien@DESKTOP-NND4UGJ ~> $ file chile.png
chile.png: PNG image data, 1920 x 1079, 8-bit/color RGB, non-interlaced
Marien@DESKTOP-NND4UGJ ~> $ file ef
ef: PE32 executable (console) Intel 80386, for MS Windows
Marien@DESKTOP-NND4UGJ ~> $ file ef.exe
ef.exe: PE32 executable (console) Intel 80386, for MS Windows
```

Figure 2.7: Using the file command on two pairs of files, with and without their extension.

Another, more powerful, tool that is also used to identify files is named Detect It Easy\textsuperscript{10}, or DIE for short. It is specialised for executable file, but it can still identify other file formats. As one can observe in Figure 2.8, it displays the compiler and linker used in the compilation process, the entropy of the file, and much more information related to the structure of the executable. If the file has been packed or obfuscated, the tool can try to find which one has been used by means of specific signatures that can be more sophisticated than magic numbers.

There is a myriad of such tools like PEiD\textsuperscript{11}, ProtectionID\textsuperscript{12}, or even Exeinfo PE\textsuperscript{13}. Those presented are nothing more than two drops in a pool of tools. Some are getting updates more frequently than others, and some are also left to die on the side. What is important is not only to grasp but also to understand how these tools actually work,

---
\textsuperscript{10}See http://ntinfo.biz/index.html.
\textsuperscript{11}See http://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/PEiD-updated.shtml.
\textsuperscript{12}See http://pid.gamecopyworld.com/.
\textsuperscript{13}See http://exeinfo.atwebpages.com/.
and to realise that, as the state-of-the-art for tools constantly evolve, it is necessary to keep looking for the sharpest tools.

2.5.3.2 Disassemblers

Disassemblers are tools that take machine code as input and produce assembly code as output. Since there are many kinds of machine languages, they are usually build specifically to work on a subset of all the languages. To be more precise, disassemblers are built to understand specific ISAs, but also file formats used to wrap code as well as relevant details which can be found in the code that is specific to each operating system. Indeed, machine code intended to be run on an Intel CPU inside a Windows operating system will not be understood by a disassembler built to work on machine code for ARM CPU.

A basic disassembly algorithm is given by Chris Eagle in *The IDA Pro Book* [10]. It takes machine code as input and it yields assembly language as output:

1. Identify where the code is located inside the executable file by parsing the binary according to its executable file format. They usually have an entry point pointer, that is the offset of the first instruction to be decoded and executed.

2. Fetch the value at the given offset in the file (starting from the entry point) and do a table lookup to match the opcode to its mnemonic. Then decode the operands according to the way the instruction is used.
3. Once an instruction has been fetched, and any required operands has been decoded, its assembly language equivalent is formatted and output as part of the disassembly listing.

4. Following the output of an instruction, we need to advance to the next instruction and repeat the previous process until we have disassembled every instructions in the file.

Two methods exist to choose which instruction is to be decoded next (step 4): Linear sweep and recursive descend. Linear sweep is the easiest method of the two because it simply decodes instruction directly below the previous one, or in other words, does a linear sweeping until it reaches the end of the code section. The main advantage of this method is that it provides a complete coverage of a program. However, it fails to determine if it is decoding data or code. Indeed, it is possible for data to be in between instructions.

Compared to linear sweep, recursive descend avoids the problem of determining whether it is decoding data by using the concept of control flow. When decoding one instruction, it chooses which one is next according to the effect the instruction has on the flow of execution of the program. It is unable to follow indirect code paths (jump, calls) which utilise some kind of lookup table.

![Diagram](image.png)

Figure 2.9: Disassembling a machine code instruction into assembly language. Image inspired by the book *Reversing: Secrets of Reverse Engineering* [17].
2.5.3.3 Decompiler

**Definition 9.** A decompiler, or reverse compiler, is a program that attempts to perform the inverse process of the compiler; given an executable program compiled in any high-level language, the aim is to produce a high-level language program that performs the same function as the executable program. Thus, the input is machine dependant, and the output is language dependant.

*Decompilation of binary programs [47]*

A fully operational decompiler is the Holy Grail of reverse engineering. If a tool can, from an executable file, generate back the original code that compiles into that executable, there would not be the need for reverse engineers but simply for software engineers. Indeed, if a tool can go backward in the compiling process up to the initial phase, all there is left to do is understanding the source code. Unfortunately, when something is too good to be true, it usually is. The state-of-the-art in decompilation is, generally speaking, not mature enough to provide that silver bullet [48].

Some of the issues that make decompiling a very complicated task have been identified by Cristina Cifuentes and K. John Gough in their paper titled *Decompilation of Binary Programs* [47] and are the following:

1. In the Von Neumann architecture, data and instructions are indistinguishable. Because decompilers are working on top of disassembler, and data and instructions can be interspersed, the errors resulting from the disassembly are passed along to the decompiler. Consequently, the decompiling cannot fall back to the original program.

2. Decompilers are usually made for decompiling executable compiled from a particular programming languages. Compilers and linkers introduce subroutines inside the executable files to, amongst other things, set up the environment before the effective instructions can be processed. These subroutines might not have been written in the programming language for which that particular decompiler is made for. It can also happen that they have been written in assembly, and might not be translatable into a higher level representation.

3. Not all the operating systems implement mechanisms to share libraries, and so solutions have been created to still allow modular programming. Shared routines from these libraries are instead embedded into the final executable in such a way that coexist with the original program (this is done by the linker). For the reason explained above, the decompiler might not be able to decompile these parts if
they are not compiled from the same programming language, or have been written directly in assembly.

Nevertheless, viable decompilers exist but with limitations. The Hex-Rays decompiler\(^\text{14}\) is a well known decompiler to a C-like pseudo code text that only works with x86, x64, ARM, and ARM64 targeted executables. It also cannot do type recovery and understanding exception handling. In Listing 2.7 and Listing 2.8, one can observe a C++ program and its decompiled version after being compiled with GCC\(^\text{15}\). The output of the decompiler is only semantically equivalent as the variable names have changed, the comments have been lost, and the overall structure is different.

```c
int fact(int n) {
    // A very useful comment
    if (n <= 0)
        return 1;
    return n * fact(n - 1);
}
```

Listing 2.7: Hand written C++ program.

```c
int __cdecl fact(int n) {
    int result; // eax@2
    if ( n > 0 )
        result = n * fact(n - 1);
    else
        result = 1;
    return result;
}
```

Listing 2.8: Output of the Hex-Rays Decompiler applied to the compiled version of the code found in Listing 2.7.

It is worth noticing that some programming languages such as Java and C# have decompilers which yield outputs that are very close to the original programs. This phenomenon can be explained by how these programming languages are compiled. They both use intermediary representations in which the ISA is still abstracted away, usually with the intent of staying platform independent. As a result, many of the information that would usually be removed by a compiler that compiles into machine code stays in the file until runtime. Obviously, once the executable starts being executed, the Java or C# compiler has to finish the job so that the CPU can understand the instructions. An example of decompiler program obtained with JD-GUI\(^\text{16}\) can be observed in Listing 2.9 and Listing 2.10. In comparison with C++ code, variable names are kept intact as well as the structure of the method. Still, the commentaries are lost as they serve no real

---

\(^{14}\)See https://www.hex-rays.com/products/decompiler/.

\(^{15}\)See https://gcc.gnu.org/.

\(^{16}\)See http://jd.benow.ca/.
purpose outside the development phases.

```java
/**
 * For a given name, generate a greeting string message.
 * @param name
 * @return the string
 */
String greets(String name) {
    String str = "Welcome, 
    str += name + ".";
    return str;
}
```

Listing 2.9: Hand written Java program.

Listing 2.10: Output of the JD-GUI decompiler applied to the partially compiled version of that code found in Listing 2.9.

### 2.5.3.4 Debuggers

Debuggers are tools that fit in the live code analysis category. They are used to observe and control the internal state and execution of a running process [38]. Compared to disassemblers, they require the executable file being analysed to be run, but at the same time, they provide more detailed information about what is happening inside such as a real time visualisation of the registers and the system memory.

Breakpoints are used to pause the execution of a process and so, it is a mandatory feature for debuggers. Without the ability to pause a process’s execution, it is impossible to observe its state changing, which is the point of a debugger. There are two kinds of breakpoints, software and hardware breakpoints. As their names suggest, the software breakpoints are implemented by the software, and the hardware ones, by the processor. Software breakpoints are implemented by replacing the instructions where the process has to stop by either a system call or an invalid instruction which will cause the debugger to take over. Hardware breakpoints are implemented on x86 architecture by special debugging registers that contain addresses. Once the process accesses one of these addresses, it will be paused by the processor, and the debugger will regain the control. Hardware breakpoints are specially useful when one wants to see when a piece of data is being accessed (a global variable, for example).

A debugger can either be attached to an existing process or start a new one from a selected application, being source-level or assembly-level, and finally be run in user or
kernel mode. These distinctions will be discussed hereunder.

If one wants to analyse a running process, it has to attach a debugger to it (a procedure that has to be done with the support of the operating system). By doing so, all the threads that compose that process will be paused, and the reverse engineer will be free to do whatever he or she wants. This can be useful when one wants to debug a process after it has been running for awhile or to analyse the changes a malware could have done to it. debuggers also allow to start a new process by selecting an executable file. In that case, the process will be paused in its entry point for the reverser to start doing its job.

Assembly-level debuggers are built on top of a disassembler and so, allows reverse engineers to debug an executable file at the level of the assembly instructions. Code-level debuggers are usually found in IDE (short for Integrated Development Environment) which are environment for programmers to be more productive when developing applications. They allow to debug directly at the source code level, which does not expose the developers to assembly instructions. In this work, the word debugger means assembly-level debugger.

As said before, the x86 ISA provides ring levels for a better separation between the operating system and the programs belonging to the users. A user mode debugger will, as its name suggests, be running in user mode, and so, will not be capable of debugging kernel level application. A kernel mode debugger can also debug user level application, but with an extended control over the operating system, as well as debugging kernel level application such as the operating system itself. To be noted that it is usually necessary to have two systems for performing kernel level debugging as to put a breakpoint in a kernel will freeze one of the two.

2.5.3.5 Strings Detecting Tools

Contrary to debuggers and disassemblers, string detecting tools do not need to have any knowledge about the structure of an analysed file, they only have to be aware of what constitutes a string. As such, these tools can be used on virtually any kind of files. The listing they produce can be useful to get a broad idea of the functionality of a program. An example would be extracting error messages that could be stored inside an executable file for they can be informative about the purpose of the application.

In a computer’s memory, characters are encoded using encodings such as UTF-8, which supports all characters defined by Unicode or even ASCII, short for American Standard Code for Information Interchange, that only handles a subset of Unicode. The strings detection tools have to be aware of these different representations to discriminate a string from the rest of what composes a file. They can also be subject to false positive. Indeed
a sequence of bytes could very well match one of the encoding while being something unrelated.

Microsoft provides a tool named Strings free of charges on its website. It can display ASCII and Unicode encoded strings with their offset (address), and it allows to filter strings that are below a certain size (three by default). Moreover, one can refine the search by specifying a starting and ending offset in the file to be analysed. An example of utilisation can be found in Figure 2.10.

![Figure 2.10](https://technet.microsoft.com/en-us/sysinternals/bb897439)

Figure 2.10: On the left, a program written in c that was later compiled in Visual Studio 2015. On the right, the listing of strings applied to the executable file produced by the program on the right. False positives are usually easy to detect as they do not mean anything.

### 2.5.3.6 PE Analysis Tools

The Portable Executable, or PE for short, is a file format used to structure executable files, object code, and DLLs on operating systems from the Windows NT family. As explained in Section 2.4.6, this structure contains the machine code as well as the meta data that are used by the operating system to load and manage the executable file. The meta data can provide a reverse engineer with information about the type of the file, the sections it contains, the debugging information left by the compiler, the resources such an icon and the manifest file, the imported functions, the exported functions, the relocation table, and the list goes on. Clearly, tools allowing to see these meta data are of great use. Hereunder it will be presented PEBrowse Professional, but it is by no means the only tool able to provide such functionalities.

![Figure 2.11](http://www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html)

Figure 2.11: On the left, a program written in c that was later compiled in Visual Studio 2015. On the right, the listing of strings applied to the executable file produced by the program on the right. False positives are usually easy to detect as they do not mean anything.

As one can see in Figure 2.11, the interface is divided into two parts. The left column contains the names of the components that compose the PE structure and the right part contains detailed view of these components. They are opened through the contextual

---


18It does not make too much sense since Unicode is not an encoding, but that is how they advertise the tool.

Figure 2.11: Screen shot of PEBrowse Professional used on a simple executable.

menu that appears once an element of the left column is right clicked. From top to bottom, the left column contains:

- **DOS Header**: For compatibility reasons, a valid MS-DOS program is the first thing one can find upon opening a file which uses the PE format. A partial view of the header can be observed on (A). One might have noticed the magic number 0x4D5A (or MZ) in the red square, which is used by the file command discussed in Section 2.5.3.1.

- **File Header**: It contains information about the file, for instance, for which system it was compiled, the number of sections, the timestamp at which the header was generated, whether it is an executable file or a DLL, and some other minor information. A partial view of the header can be observed in (B).

- **Optional Header**: This header is not optional. Inside it can be found the version of the linker used to make the file, the entry point of the program, a checksum of the file, and much more information.

- **Sections**: As the name suggests, this part contains the different sections that compose a program. “.text” contains the machine instructions, “.data” contains
global data, ".rdata" has the same use with the difference that it is read-only, 
".reloc" contains the relocation table, ".rsrct" contains the resources of the program such as the manifest that tells the operating system if an elevation of privileges is necessary. The tool also contains a disassembler, and a view of its output can be observed in (C).

- **Imports**: Here it can be found the DLLs which export functions used by the file, as well as the function names.

- **Resources**: It is a shortcut to get to the resources stored in the file.

- **Debug**: It is a shortcut to get to the debugging information which could come with the file.

The PE format is a very complex subject. For more information on the topic, see the article of Matt Pietrek published in the MSDN Magazine [49].

### 2.5.3.7 Tracing Tools

Tracing is a technique that allows someone to understand what is happening in a software system\(^{20}\) by tracing the execution of its processes. A tool able to perform tracing is called a tracer. An interesting property of this family of tools is that, they do not require the reverser to dive into assembly code. Indeed, their only purpose is to record the occurrences of certain events that are triggered by the probes they disseminate. These events can either be printed on the screen or saved in a file for further analysis, and the probes can be installed inside the operating system’s kernel or any software application.

It is important to differentiate logging from tracing for they do not operate at the same level of complexity. On one hand, logging is used for high level analysis of infrequent events such as networking failure or database accesses. On the other hand, tracing is used at a very low level to monitor events such as system calls and library calls.

**Strace**\(^{21}\) is an open source tracer developed for Linux and provided in most distribution to intercepts system calls made by processes as well as signals received by processes, with the support of the kernel. It can also trace the child processes as they are created by the main process, trace interactions with the kernel, and provide options to specify what kind of system calls and signals must be logged. **Ltrace**\(^{22}\) is also a tracer but with the particularity that it can only log calls made to shared libraries. It then does not require support from the kernel and provides more readable outputs. In Figure 2.12 can be observed the output of ltrace on a simple “hello world” program.

---

\(^{20}\)A software system is the set of programs which are running on a specific hardware system.

\(^{21}\)See [http://man.he.net/man1/strace](http://man.he.net/man1/strace).

\(^{22}\)See [http://man.he.net/man1/ltrace](http://man.he.net/man1/ltrace).
2.5.3.8 Monitoring Tools

Monitoring tools are similar to tracing tools for they provide information about the behaviour of processes. The difference is that they yield outputs that are more high level than, for example, a listing of all the system calls made by a specific process. Because communication with the operating system is necessary to make use of a computer, and since tracers are used to analyse these exchanges, monitoring tools are usually built on top of tracers. As it might be expected, they do not require the reverser to work at the level of assembly code either.

The Windows Sysinternals Suite\textsuperscript{23} is a collection of tools made freely available by Microsoft to help with managing, troubleshooting and diagnosing Windows systems and applications. Some of its tools will be discussed hereunder.

- **Handle**: For every process in the system, this tool gives its list of open handles. A handle is a reference to an open file, which can also be a registry key.

- **ListDLLs**: As its name suggests, this tool is used to list all the DLLs that are loaded into a process.

- **Procmon**: Short for Process Monitor, it is a combination of legacy tools called respectively Filemon and Regmon (File and Registry Monitor). It displays in real time file system, registry and process/thread activity while providing very useful filtering capabilities.

- **TCPView**: This tool lists all the TCP and UDP connections made by the processes of the system. For each connection, it is displayed the protocol, local and remote addresses, local and remote ports, the state of the connection, and statistics about the exchanged packets.

- **Sysmon**: Short for System Monitor, it is a service and device driver that does not have to be manually restarted across reboots. It logs information about system activities such as process creation, changes to file, and network connection.

- **Procexp**: Short for Process Explorer, it is a more complete version of the task manager provided by default on Windows. It shows the currently active processes,

and for each of them, the open handles, the loaded DLLs, and the memory mapped files, while providing the other functionalities found in the original task manager. Compared to the other tools discussed above that provide the same capabilities, procexp has an intuitive graphical interface that makes it easier for these activities to be monitored.

2.5.3.9 Virtual Machines

A virtual machine could be seen as a computer running inside another computer. The host computer is usually a concrete computer that has a CPU which provides functionalities for virtualisation, and the guest computer is the computer running on top of the host computer. This can be observed in Figure 2.13, where normal applications, such as a browser, are seen sharing the system with a virtual machine.

![Virtual Machine Diagram](image)

Figure 2.13: A virtual machine living in an operating system. Image inspired by the book *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software* [38].

A virtual machine can be useful depending on the kind of executable that is being analysed. If someone running on Windows system wants to perform live debugging on an executable compiled for Linux, he or she might want to make use of this technique. Another situation that justifies the need of virtualisation is when dealing with malware applications. Because these programs can have disastrous effects on the system it is being analysed on, and on the neighbouring systems as well, it is greatly advised to isolate that system from the rest of the world.

An interesting functionality of virtual machines is the possibility to save the current state of a virtualised system to restore it later. The saved state is usually called a snapshot. An example of how this technology could be used would be to take a snapshot
of a clean system, run a malware, analyse the damages/changes it has done, and then finally roll-back to the clean state to analyse another malware.

VMware Workstation\(^{24}\) provides another feature called record/replay that can be useful to speed up debugging sessions. Once activated, VMware will start recording through its virtualisation layer the complete execution behaviour of the applications being executed inside the virtual machine. This recording allows the machine to go back in time to replay the same exact behaviour, over and over again [50]. Conceptually, it is equivalent to a system wide “undo”. If, for example, a reverse engineer enters a function that never ends, he or she can either restart the debugging session or replay the recording until a little bit before jumping into that function.

### 2.5.3.10 Memory Scanning Tools

These tools are used to scan the memory of a process for specific variables using filtering rules that provide an iterative refinement. Examples are often worth a thousand words, so let’s explore a simple scenario. One could find itself stuck in a level of a very challenging video game. Instead of applying the well-known fail and retry approach, the lazy gamer can use a memory scanning tool to identify variables of interest such as the life or score counter with the intent of applying beneficial modifications.

In our previous scenario, the challenge of beating the game has been replaced by finding these variables. To do so, the tool has to first identify all the possible variables present in the process, and then to provide the possibility to rescan for specific changes in these variables. Those who did not changed in that specific way are filtered out. When the player takes damage, the variable holding the amount of life left will in most case decrease. One can try to find its location by taking damages, scanning for decreasing variables, and repeating these operations until one variable is left.

When using a Linux system, one can use scanmem\(^{25}\). It provides a means to both locate and modify a variable in a running process, but no graphical interface is provided directly. On Windows, there is the famous Cheat Engine\(^{26}\) that also provides a means to locate and modify variables, but also comes with a useful graphical interface that makes the research easier, and consequently faster. It can be observed in Figure 2.14.

### 2.5.3.11 Hex Editors

Hex editors are to programs what text editors are to text files. They provide a means to edit files of any kind through their hexadecimal representations instead of the textual...
Figure 2.14: Cheat Engine used on The Binding of Isaac: Rebirth.

one, if it exists. It can be observed in Figure 2.15 HxD\textsuperscript{27} tool, a free hex editor running on all versions of Windows starting from 95. The main window contains three columns that can be identified by the spaces that separate them. On the left, one can see the starting addresses of the 16 byte arrays that are found in the middle column. On top of that column are given the offsets that have to be added to the array address to get the address of each byte. The right column simply shows a textual representation of the bytes using a specific encoding. Here, ANSI is used.

2.5.3.12 Visualisation Tools

Given enough time and resources (such as man power and tools), any reverse engineer can potentially extract what he or she is looking for from a binary file. Life being what it is, ephemeral, we all have a limited amount of both. Visualisation tools shine at providing information about the overall perspective of a file in a way that requires little to no time to be understood. Consequently, it can greatly speed up the reversing process. In this section, two kinds of tools will be discussed. The first one does not care

\textsuperscript{27}See \url{https://mh-nexus.de/en/hxd/}.
The entropy measures a system’s disorder. The higher is the value, the less ordered is the system. A sequence of the same characters has then little entropy, whereas a sequence made of all the characters that can exist on that system will have the highest entropy. Aldo Cortesi, a New Zealander security consultant had the idea of computing the entropy of a file using the Shannon entropy [51] over a sliding window to make the task of finding compressed data/cryptographic material inside executable files easier. They indeed have higher entropy levels than regular data such as strings and assembly code. To display the information in an intuitive way, he used the Hilbert curve [52] for it gives a mapping between one dimensional and two dimensional spaces that preserves locality. His tool\(^{28}\) can be used online, directly on his website. In Figure 2.16, it can

---

\(^{28}\)See http://binvis.io/.
be observed the result of applying this analysis to TCPview. To be noted that entropy is not the only information that can be displayed with that tool. The byteclass colour scheme gives a blue colour for character, and different colours for non textual data, as seen in Figure 2.17. There are two other schemes at the time of writing, one can refer to the author’s website for more information.

A control flow graph (CFG for short) is a representation of all the paths a program can take during its execution. For example, conditional statements such as an *if–then–else* split the path of execution at least into two trails. A CFG is very useful as it shows these branches in a directed graph and gives insight on how the program can unfold. A trained reverser can even easily identify specific constructs simply by looking at the structure of the graph. A CFG generated with IDA\(^29\) can be observed in Figure 4.1. A creative reader could find the totally useless but quite remarkable tricks to turn a control flow graph into a grey scale image interesting. This idea has been developed by Christopher Domas with the goal of deterring potential reverse engineers by “crushing their souls”\(^30\).

Another more useful work done by Christopher Domas is the interactive binary visualisation tool ..cantor.dust..\(^31\) presented in various security conferences. It uses the ideas of Aldo Cortesi and Gregory Conti [53] to visualise information in a graphical

---

\(^29\)A disassembler and debugger which provide a plethora of functionalities. See [http://www.hex-rays.com/](http://www.hex-rays.com/).

\(^30\)See his DEF CON presentation: [https://github.com/xoreaxeae/REpsych](https://github.com/xoreaxeae/REpsych).

\(^31\)See [https://sites.google.com/site/xxcantorxdustxx/visual-re](https://sites.google.com/site/xxcantorxdustxx/visual-re).
Figure 2.18: A control flow graph generated by IDA. If-then-else constructs are easily identifiable.
way, but also introduces an automated mechanism for classifying different regions of a file according to the type of information they contain using statistical methods. The annotated output can be observed in Figure 2.19. Sadly, at the time of writing, the tool has not been released to the public and the author has been silent on the subject for a rather long period of time. For more information, one should watch one of his presentations entitled “The Future of RE Dynamic Binary Visualization”.

![Figure 2.19: Annotated output of ..cantor.dust.. Image from the presentation.](image)
2.6 Obfuscation Techniques

**Definition 10.** Obfuscation is the action of making something obscure, unclear, or unintelligible. 

*Oxford Dictionaries [54]*

Reverse engineering is inherently complicated for it requires the reverse engineer to think backward to rediscover buried knowledge, know-how and details from artefacts. This process can be made further more complicated by the artefacts’ engineers through the use of obfuscation, a family of techniques that belongs to the anti reverse engineering techniques. The family of obfuscation techniques provides means to make programs more opaque to scrutiny by transforming them into new programs that have the same computational effect while being harder to analyse [36]. Anti reverse engineering is a broader family for it also contains techniques that aim, for example, at thwarting reverse engineering tools, detecting virtual machines, and so on.

When obfuscation is applied to programs, a harsh reality has to be understood. There is no such thing as a protection which provides a fully opaque filter. Programs are usually being shipped in executables files that are understandable either by the hardware or specific software, and in order to provide a specific behaviour, these executables have to lay a detailed description of that behaviour. One could make the comparison with a blueprint of any kind of engineered artefact. Making it hard to read won’t prevent someone motivated enough to make sense out of it, but it can wear some reverse engineers out by making them give up and moving on if the process is slow and painful enough. Not any problem has a neat solution, and preventing reverse engineering is one of these problems. As hinted above, security through obscurity, a solution that is widely discouraged, is the only alternative that can mitigate the risks of having a program being reverse engineered.

According to the book *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation* [36], the obfuscation techniques can be sorted into two categories: Data-based obfuscation and control-based obfuscation. Individually, they do not provide much obscurity. It is only when they are applied together that the reverse engineering process becomes more challenging. The analogy made by Jakubowski et al [55] about round based cryptography and the iterative application of obfuscation techniques illustrates very well this idea: “A cryptographic algorithm’s round is made of basic arithmetic operations (addition, exclusive or, etc.) that perform trivial transformations on the inputs. Considered individually, a round is weak
and prone to multiple forms of attacks. Nevertheless, applying a set of rounds multiple
times can result in a somewhat secure algorithm. That is the objective of an obfuscator.
The objective of the attacker is to discern the rounds from the global obfuscated form
and to attack them at their weakest points.”. The remaining of this section will be
dedicated to describe known obfuscating techniques that could make the rounds of a
practical obfuscator.

2.6.1 Data-based Obfuscation

2.6.1.1 Constant Unfolding

Constant folding is a compiler optimisation technique that consists of evaluating con-
stant expressions at compile time and replace the constant expressions by their val-
ues [31]. When writing an expression such as $\tau = 3.14 \times 2$, one can easily see that
reducing it at compile time into $\tau = 6.28$ will not alter the semantic of the original
program while increasing the runtime performance.

Applying constant unfolding at the assembly level is not always as straight forward
as it could be in a higher level programming language. For example, in the situation
where we want to unfold an expression that sets the content of the $ax$ register to $0x200$,
we could use the following expression:

\begin{verbatim}
  mov ax, 100h
  mov bx, 100h
  add ax, bx
\end{verbatim}

Listing 2.11: Example of constant folding.

Compared to the folded expression, this one will not only change the content of the
$bx$ register, but also the content of the $eflags$ register. More precisely, it will change the
content of the overflow flag, the sign flag, the zero flag, the adjust flag, the carry flag,
and the parity flag [34]. One has thus to be careful when applying constant unfolding
in the context of an assembly language that has side effects for they can change the
semantic of the program.

Countering this technique is pretty straight forward, so applying the constant folding
optimisation will reduce the expressions to constants.

2.6.1.2 Data-Encoding Schemes

One could encode the values of a program stored in the variables and the constants
while adding an encoding and decoding function to allow to manipulate them during
run time. A simple example would be to have an encoding function that adds the value
$x$ and a decoding function that subtracts the same value $x$ of its argument. The major flaws of this technique are that the encoding and decoding functions have to be inside the program, that the variables and constants have to be decoded and thus exposed when used, and lastly that a constant folding optimisation would discard it.

**Definition 11.** A homomorphism is an operation-preserving mapping between two algebraic structures.

*Oxford Dictionaries [54]*

Homomorphism could be a better solution for it does not make decoding variables mandatory to manipulate them and so it does not expose them. Homomorphism is an operation-preserving mapping between two algebraic structures. To better illustrate this concept, let’s take two groups, $A$ and $B$, and two operations, $+_A$ and $+_B$, belonging to the group mentioned in the subscript. The function $f$ is an homomorphism between the sets belonging to $A$ and $B$ if $f(x +_A y) = f(x) +_B f(y)$. It is said that this notion can be generalised to arbitrary algebraic structures such as rings and, for example, the addition and multiplication operators. For more information on this topic, see the work of Zhu and Thomborson [56].

### 2.6.1.3 Dead Code Insertion

Dead code is the name given to instructions whose results do not affect the behaviour of the program [60]. They could either never be executed, or have no effect in the current computation. Since they only make the executable files bigger in size, most compilers will always try to eliminate the high level instructions which result in dead code once compiled. This method is called dead code elimination. See *Advanced Compiler Design and Implementation* [61] for more information on the topic.

Dead code insertion is the exact opposite of what has been explained above. One can insert instructions that either do not alter the behaviour of the code or are applied to dead registers. The main goal of this technique is to make the code harder to read, forcing the reverser to decide whether an instruction is meaningful or not.

An example of dead code insertion is illustrated in Listing 2.12. The semantic of the function is to sum the two parameters that it is applied to. While the two last lines are necessary to provide the desired outcome, all that comes before is not. The variable $w$ is never used, and the conditional branch will never be taken. If compiled with the dead

---

32 A dead register is a register which is not used.
code elimination optimisation activated, they would not make it to the machine code.

```c
int add(int x, int y) {
    int w = 50;
    if (false) {
        printf("dead\n    code");
    }
    int result = x + y;
    return result;
}
```

Listing 2.12: Example of function with dead code.

In the example, the variable \(w\) is said to be dead for it has no effect on the computation. On the other hand, the variable result is said to be live since it carries the result of the addition. The same idea can be applied to registers, some partake in the computation, some others don’t. This notion is important because when an obfuscator adds dead assembly code into a program, it might have to avoid using live registers to prevent altering the semantic of the program.

### 2.6.1.4 Arithmetic Substitution via Identities

When dealing with mathematics, there usually are plenty of different ways to solve a single problem. Some are easier to understand than others, and as expected when trying to obfuscate programs, the harder the better. Arithmetic substitution is about substituting mathematical expressions with semantically equivalent but syntactically different expressions using identities. One of such identities is the following: Instead of simply adding 1 to a register when needed, one could write an expression that xor the value of a register with 0XFFFFFFFF and apply the unary minus operator to the result. For the binary value 0011 (3), xoring it with itself gives 1100 and negating the result yields 0100 (4). To be noted that this substitution only works in a system using the two’s complement signed number representation.

Below are listed a few identities from the book *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation* [36]. The symbol \(~\) is the not operator, \(rotate\{left, right\}(x, y)\) performs a rotation of \(y\) bits on \(x\) in the chosen direction, \(nb\_bits(x)\) returns the number of bits that makes a value \(x\).

- \(-x = \sim x + 1\)
- \(x + 1 = - \sim x\)
- \(x - 1 = \sim -x\)
\[ \text{rotate}_\text{left}(x, y) = (x << y) \mid (x >> (\text{nb}_\text{bits}(x) - y)) \]
\[ \text{rotate}_\text{right}(x, y) = (x >> y) \mid (x << (\text{nb}_\text{bits}(x) - y)) \]

The Information Security Group of the University of Applied Sciences and Arts Western Switzerland has developed an obfuscator for the LLVM Intermediate Representation, or IR for short, language [57] that uses different substitutions such as the following:

- \( b \& c \) becomes \((b \oplus \sim c) \& b\)
- \( b \mid c \) becomes \((b \& c) \mid (b \oplus c)\)
- \( a \oplus b \) becomes \((\sim a \& b) \mid (a \& \sim b)\)
- \( a = b + c \) becomes \(r = \text{rand}(); a = b - r; a = a + b; a = a + r\)

Just like using a single obfuscation technique to harden your program is not very useful, applying only one identity will yield poor results. One will instead want to apply many of them, possibly on the result of other permutations. Creativity and the overhead gained from expanding simple operations into more complex ones are the only limitations of this obfuscation technique. Combining the substitution will make it harder for a reverser to understand the underlying logic but will have an impact on the overall performance and size of the program.

2.6.1.5 Pattern-Based Obfuscation

Pattern-Based Obfuscation is, in a way, similar to the arithmetic substitution presented in Section 2.6.1.4. Instead of substituting mathematical operations, this technique consists of substituting a set of adjacent instructions into another set of instructions that has an equivalent semantic. One example can be observed in Listing 2.13 and Listing 2.14, where the \textit{jump} instruction is said to be equivalent to pushing the destination address on the stack and calling the \textit{ret} instruction right after. Because the semantic of \textit{ret} is to \textit{pop} the first at the top of the stack and \textit{jump} to that value, the semantic of \textit{jmp} is replicated.

\begin{verbatim}
jmp  addr
\end{verbatim}


\begin{verbatim}
push addr
ret
\end{verbatim}

Listing 2.14: Program equivalent to the one in Listing 2.13. Semantic equivalence preserved.

\footnote{LLVM is a collection of modular and reusable compiler tool chain technologies. See \url{http://llvm.org/} for more information.}
Other examples could include the push and pop operations, which take or put a value on the stack and increase or decrease the value of the stack pointer. Listing 2.15 and Listing 2.16 show the former identity, Listing 2.17 and Listing 2.18 show the later identity.

```
push eax
```
Listing 2.15: Program equivalent to the one in Listing 2.16. Semantic equivalence preserved.

```
pop eax
```
Listing 2.17: Program equivalent to the one in Listing 2.18. Semantic equivalence preserved.

```
mov dword ptr [esp], eax
sub esp, 4
```
Listing 2.16: Program equivalent to the one in Listing 2.15. Semantic equivalence preserved.

```
mov dword ptr [eax], esp
add esp, 4
```
Listing 2.18: Program equivalent to the one in Listing 2.17. Semantic equivalence preserved.

It is once again important to note the side effects an instruction can have when using it as a substitution. In the first example, where the jump instruction is replaced by a push and ret, no flags are affected and so they are both semantically equivalent. On the other hand, the push and pop equivalents are not semantically equivalent for add and sub change flags.

### 2.6.2 Control-based Obfuscation

#### 2.6.2.1 Inline and Outline Expansion

Inline expansion is a compiler optimisation which consist of replacing specific function invocations by their body, which are altered beforehand to take into account the new way the parameters and the return value will be passed [31]. One advantage of this technique is that it makes the code faster by removing all the machinery that is necessary when calling a function. The main disadvantage is that the code will grow in size since the body will be duplicated as many times as it is called.

Outline expansion is simply the opposite operation, it consists of extracting a piece of code, turning it into a function, and adding a function call for that function. The advantage and inconvenient of using this technique are obviously the opposite of the ones for inline expansion.

Using inline and outline expansion the right way can greatly degenerate the call graph of the application. A call graph is simply a graph which shows what functions a function
might be calling. A simple example can be observed in Figure 2.20. By doing so, the graph will be greatly harder to read, and so to reason about.

![Figure 2.20: Part of the call graph of the Microsoft Resource Compiler generated with IDA pro.](image)

### 2.6.2.2 Removing Sequential and Temporal Locality

A basic block of code is a sequential listing of code which does not have any “in-branches” other than its entry point and no “out-branches” except its exit point. When a compiler encounters such a block, it will generate one continuous block of instructions. This property, called sequential locality, makes reverse engineering easier for one can simply read the instructions of a block in a sequential order, without caring about branches, to get a gist of its purpose.

Compilers will also put temporally related blocks next to each other. For example, if the exit point of a block is the entry point of another block, they will be stored side by side in the executable file. This property is called the sequential locality of temporally related code [36]. These two properties are important for a performance point of view because of how CPU caching is done, that is by decoding the next logical instructions in advance in hope that they will indeed be part of the execution path.

This technique, as its name suggests, consists of modifying the machine code so that these two properties are not satisfied anymore. By inserting unconditional branches inside the blocks of code and displacing the temporally related block, a reverse engineer will have a harder time reasoning about the code. This technique will not be of any use against automated deobfuscation tool.
2.6.2.3 Opaque Predicates

**Definition 12.** A predicate $P$ is opaque if a deobfuscator can deduce its outcome only with great difficulty, while this outcome is well known to the obfuscator.

A taxonomy of obfuscating transformations [58]

The idea behind opaque predicates (boolean expressions) is that, whether they will be evaluated to true or false will not clear from the point of view of an attacker until being evaluated, while always evaluating to the same value. One usage is to use such a predicate in a conditional branch, turning it into an unconditional branch. Of course, that is information a disassembler will not have since it will not reduce the expression. As a result, another, dead, branch will be added into the control flow graph of the application. From there, one could either put junk code or keep complexing the control flow graph by adding branches.

A simple example of opaque predicate would be comparing two different constant numbers and using the jump not equal instruction to branch to the valid continuation of the program. Another, more complex, example could be made by using the greatest common divisor (GCD) algorithm. Being an associative binary operator, one could choose a set of integers and recursively pick and apply numbers to the result of the previous application of the GCD algorithm. When the set gets fully consumed, checking the parity of the result will give a true or false answer. Pseudo code for this algorithm can be found in Listing 2.19.

```plaintext
let s := {...}
let accumulator := random_get(s)

while s != empty
    accumulator := gcd(accumulator, random_get(s))
end while

if (accumulator % 2 == 0)
    ...
then
    ...
end if
```

Listing 2.19: Pseudo code for an opaque predicate based on the greatest common divisor algorithm.
In *Practical Reverse Engineering* [36] is proposed a variant that works as follows: Instead of having an opaque predicate that always evaluates to the same value, one could make it evaluate randomly to true or false while having the two branches being semantically equivalent. As they would pass the control flow to the same instruction once finished, it would give rise to a diamond shaped control flow graph.

For more information on the topic, one might be interested in reading these two articles: *Manufacturing cheap, resilient, and stealthy opaque constructs* [59] and *A Taxonomy of Obfuscating Transformations* [58].

### 2.6.2.4 Interleaving Function’s Body

As its name suggests, this technique consists of taking functions’ body and splitting them into fragments that are then interleaved and connected with unconditional jumps composed with opaque expressions. These expressions will have to reduce to the memory addresses of the next fragment. That way, it is not obvious which fragments belong to which functions. This technique will make it harder, both for a reverse engineer and a tool, to make sense of the code. An example where two functions are interleaved can be observed in Listing 2.21.

```c
function_1() { 
    function_1_step_1
    function_1_step_2
    function_1_step_3
}

function_2() { 
    function_2_step_1
    function_2_step_2
    function_2_step_3
}
Listing 2.20: Two functions and their body.
```

```c
function_2_step_1  
  jmp opaque_expression
function_1_step_3  
  ret
function_1_step_1
  jmp opaque_expression
function_2_step_2
  jmp opaque_expression
function_2_step_3
  ret
function_1_step_2
  jmp opaque_expression
```

Listing 2.21: Interleaving of the two functions found in Listing 2.21. Inspired from the book *Reversing: Secrets of Reverse Engineering* [17].
2.6.2.5 Processor Based Control Indirection

This technique consists of obfuscating the two most obvious ways branching is done in machine code, that is with the \textit{call} and the \textit{jmp} instructions. The interest is that, besides the fact that the listing produced by a disassembler will be harder to understand for humans, many tools will not recognise the potential branching, and so won’t be able to provide as many details as they usually would.

Whenever a disassembler discovers a \textit{call} instruction, it will interpret the address applied to the instruction as the entry point of a function. Since functions are made to give back the control once finished, the disassembler will also assume the presence of the \textit{ret} instruction, signalling the end of the function. Replacing these two instructions by semantically equivalent sequences of instructions will then, for example, prevent the generation of the control flow graph.

In Listing 2.22 can be seen an example where \textit{call} has been replaced by three instructions: One to get the instruction pointer (\textit{ip}), one to increment the \textit{ip} value to point toward after the function call, and finally a jump to the \textit{foo} function. When loaded into IDA Pro, the \textit{foo} function will not appear in the call graph nor in the control flow graph.

\begin{verbatim}
getEip PROC
    mov eax, [esp]
    ret
getEip ENDP

foo PROC
    invoke MessageBox, NULL, addr MsgBoxText,
    addr MsgBoxCaption, MB_OK
    ret
foo ENDP

start :
    call getEip ; This
    add eax, 06h ; is
    push eax ; replacing
    jmp foo ; call foo
    ... ; getEip + 06h
end start
\end{verbatim}

Listing 2.22: MASM code which hides the function call to \textit{foo} by using a set of instructions with an equivalent effect.
For the *jmp* instruction, it is fairly easy to emulate it with a *push* and *ret* as explained in Section 2.6.1.5. It is also feasible to use *call* on the landing address and popping the return address that will have been pushed by the *call* instruction. By doing so, the control flow graph will get polluted.

### 2.6.3 Combining Data and Control Flow Techniques

#### 2.6.3.1 Junk Code Insertion

This technique presented in the book *Practical Reverse Engineering* [36] uses the dead code insertion technique and the opaque predicate technique to try to thwart the disassembler by making it follow specifically crafted branches that will not be followed by a CPU. The branch can contain simple dead code or jumps to invalid addresses, which would make the disassembler desynchronize.

A simple example written in MASM can be observed in Listing 2.23. In the situation where eax starts with the value zero, adding it to itself will never cause overflow. As a consequence, the conditional jump will always be taken, skipping the junk code. The disassembler, not being aware of this, will think that this function is recursively calling itself and that the *ret* instruction is part of another function. When loaded in IDA Pro, the control flow graph in Figure 2.21 appears.

```
start:
    add eax, eax
    jno end_junk
    jmp start
    ret
end_junk:
    invoke ExitProcess, NULL
end start
```

Listing 2.23: Inserting junk code to trick the disassembler into believing that the function is recursive and that *ret* is part of another function.

#### 2.6.3.2 Control Flow Graph Flattening

Control flow graph flattening is a technique which consists of transforming a source code into another source code that will produce a flattened CFG once statically analysed. A flattened CFG is a graph which has one dispatcher node that is connecting every other nodes, and every other nodes are connected to the dispatcher node to give it back the control flow once over. A reverse engineer, when faced with a flattened CFG, will not be able to follow the branches easily, making it harder to perform static analysis. On
the other hand, the extra branching will have a cost on the overall performance of the application.

Below is given a simplified algorithm for CFG flattening. It has been proposed in the article titled *Obfuscating C++ Programs via Control Flow Flattening* [62].

1. Break a function’s body into basic blocks and put them next to the other. Note that before this operation, the blocks were not at the same level of nesting.

2. Encapsulate the blocks in a switch-like construct, where each block has its own case/break separator.

3. Wrap the whole construct in a loop.

4. Add a state variable that gets updated at the end of every basic block. This variable is used by the construct to find the next basic block to be executed.

An example of CFG flattening can be observed in Figure 2.22, which is the result of applying the algorithm on the code found in Listing 2.24. One should pay attention to how the while loop from the original code has been rewritten into two basic blocks: One for the body and one to check whether the predicate still holds or not.
```c
int main(int argc, char **argv) {
    int n = 50;
    int r = 1;
    while (n != 0) {
        r = r * n;
        n = n - 1;
    }
}
```

Listing 2.24: Simple code to compute the factorial of 50.

![Control flow graph of the code from Listing 2.24 once flattened. Example inspired from Obfuscating C++ Programs via Control Flow Flattening [62].](image)

2.6.3.3 Virtual Machines

It has been said before that some languages are first compiled into an intermediate representation (IR) to provide, amongst other things, better portability. Whenever a file containing such code needs to be executed, a just-in-time (JIT) compiler will dynamically compile the intermediate instructions into machine code and let the CPU execute them.

For a reverse engineer to analyse such language, he or she needs to know the semantic of the instructions composing the language combined with the architecture of the virtual machine over which the instructions are being interpreted. For well known IRs such as
the Java Bytecode, one can simply refer to the official documentation, but when both
the language and the architecture are kept secret, the analysis suddenly turns into a
tedious task.

Contrary to mainstream uses of virtual machines, obfuscating virtual machines will
embed the JIT compiler (or interpreter) inside the executable file, next to the interme-
diate instructions. The interpreter can obviously not be written using the IR for the
CPU would not be able to make sense out of it. Whenever such executable is launched,
the control flow is given to the interpreter which will proceed to read and evaluate the
intermediate instructions. As a result, the interpreter is the only part of code that can
be statically analysed.

The disadvantages of this method are that it is very complicated to engineer a virtual
machine, and the performance of the application will be greatly diminished.

2.6.4 Other Anti Reverse Obfuscation

As explained previously, it is a tedious process to preserve the semantical equivalence
when tinkering with assembly code. Moreover, obfuscating a program has a cost that
one might not be keen to pay, or simply cannot afford. The techniques presented in the
remaining of this section also aim at slowing down the reverse engineering process, but
in a way that does not require rewriting code. They thus do not fall into the obfuscating
family any more but are worth being acknowledged nonetheless. To be noted that, once
again, using only one technique will not produce a good protection. As the national
motto of Belgium says, “unity makes strength”.

2.6.4.1 Removing Symbolic Information

Symbolic information are pieces of information that can be found in binary files such
as executables and dynamic-link libraries (DLLs). According to their nature, they can
help a reverse engineer carrying out his or her task with very little effort and so must
be taken into consideration. The amount of information found in a file varies according
to its type and the compiler used to produce the file. The two most verbose cases are
the import/export tables and when dealing with partially compiled code such as Java
bytecode.

An executable using the PE format presented in Section 2.4.6 will contain an Import
Address Table (IAT). In it, it could be found the name of the modules that contain
functions needed by the executable, as well as the name of the functions or their ordinal
inside the module. Names and ordinals are equivalent for they uniquely identify one
function, but they differ on how they identify it. The name is a textual representation
while the ordinal is just a number. It is obvious that more can be inferred from a name
since they are usually chosen to describe their behaviour. In Figure 2.23 can be observed the import table of a program that checks if a debugger has been attached to it. It would not have been as obvious if “isDebuggerPresent” had been replaced by, let’s say, “6”.

Figure 2.23: Import table of a program that checks for the presence of a debugger.

For the IAT to be filled with addresses pointing to another module at runtime, the other module has to specify which functions it is exporting as well as their relative addresses in the module at compile time. This is done with the export table that is also found in the PE header. Once again, they can be listed either by ordinals or by name. If the module decides to export its functions by ordinal, they will have to be imported by ordinal. As such, the verbose identified will be replaced by something more discreet.

The other case mentioned above is when dealing with partially compiled code. Because the names declared by the programmers are most of the time kept intact, instead of turning them into addresses, it is possible to go back to a version of the code that is strongly similar to the original one [17]. In this situation, the symbolic information cannot be removed. Instead, it can be changed for something less informative. An example would be to replace “isDebuggerPresent” by “fct_15”.

2.6.4.2 Anti Debugging

A debugger is one of the key tools used to carry out reverse engineering, and that makes it a target of choice. With reverse engineers relying on their tool to perform their
analysis, confusing the tool would mean confusing the reverse engineer that is sitting at the other side of the tool. This can be achieved in many ways, for example, by exploiting vulnerabilities from the debugger, by changing the behaviour of the program or even by stopping the execution of the program. The book *Practical Malware Analysis* [38] proposes many techniques, two of which are presented below.

To set breakpoints, debuggers replace the line at which the breakpoint has to be inserted by the INT 3 instructions. One could spawn a thread with the sole purpose of looking for that specific instruction in the sections of the process that contains instructions.

Another way of finding INT 3 instructions is to perform checksum on the sections containing instructions. These two protections can be beaten by using hardware breakpoints instead of software ones.

**2.6.4.3 Confusing Disassemblers**

In Section 2.5.3.2 it has been explained how disassemblers operate to translate machine code into assembly code. It has also been stressed how important a disassembler is to perform reverse engineering for it constitutes the foundation of many other tools such as debuggers and decompilers. As a consequence, it is not uneasy to understand the benefit of embedding anti disassembler protections in one’s application. Their goal is to desynchronise the disassembler from the flow of instructions, which will result in an incorrect listing of assembly code.

Linear sweep, one of the two methods disassemblers use to decide which part of memory is to be decoded next, works by simply sweeping linearly through the code section. For IA-32 processors, instructions are not all of the same size. Inserting specifically crafted code after a conditional jump can thus desynchronise the disassembler and lead to an incorrect listing. An example can be observed in Listing 2.25 and Listing 2.26.
In the example, the two disassemblers correctly translate the first line, an unconditional jump. The second instruction, on the other hand, is not translated correctly by WinDbg. The cause of this problem is that WinDbg uses the linear sweep method, and so mistakes the data for the beginning of the next instruction. Moreover, the third instruction, `push 0`, never appears in the listing of WinDbg. This is because the bytes that make this instruction have been consumed to generate the second instruction of the listing. Past that point, WinDbg resynchronises correctly and gives the same result as IDA Pro.

As one might have deduced, IDA Pro uses a recursive traversal algorithm, which instead of linearly translating instruction, follows the control flow of the code whenever it encounters a branch or jump instruction. These kind of disassemblers will not fall for the techniques based on the one presented above, but are not exempt of flaws for the cause. Using opaque predicates (see Section 2.6.2.3), one can confuse a recursive traversal disassembler with disassembling data.
Chapter 3

Equational Reasoning on x86 Assembly

3.1 Algebra of Program

One might reason about an equation to decide whether it is a consequence of an equa-
tional system (i.e. a set of equations), but that is not the only application of equational
reasoning. In the context of computer science, and more precisely, in the context of pure
functional programming languages, one could see his or her code as an equation made of
smaller equations that have been glued together with special operators. It would then
be possible to apply rewriting rules to change the representation of the code [64]. One
example would be to replace a function call by its body, or the body by a function call.
A simple example of this can be observed in Listing 3.1. One could go further by adding
a function making a call on any of the three functions, with concrete parameters, and
reduce it to a single integer value.

\[
\begin{align*}
\text{square} & : \text{Integer} \rightarrow \text{Integer} \\
\text{square} x &= x \times x \\
\text{pythagoras} & : \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{pythagoras} a b &= \text{square} a + \text{square} b \\
\text{--- Replacing the square definition by its body} \\
\text{pythagoras'} & : \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{pythagoras'} a b &= a \times a + b \times b
\end{align*}
\]

Listing 3.1: Example of equational reasoning in Haskell.

This algebra of programs (also called equational reasoning) is a technique that has
emerged from the functional programming world in response to the problem of proving
program correctness. Its strength resides in the fact that, contrary to most formal program solving methods, an average programmer is able to prove the correctness of his or her (functional) programs without requiring to master a panoply of advanced mathematical and logical concepts [63]. Indeed, a programmer would use his or her programming knowledge of the language to derive proofs, just like one would do with algebraic proofs.

For this algebra of programs to work, it requires the underlying language to be purely functional, that is, to not allow variables to mutate. Indeed, if they were allowed to change their states, the order of evaluation (reduction) would have an impact on the semantic of the code. An implication of this property is the gain of what is called referential transparency. It can be informally explained as $f(x) = f(x)$: A function, when applied to the same parameters over time, will always give back the same result. A consequence of this property is that every function applications to the same parameters can be replaced by their result. This property will not hold on for imperative languages. Indeed, being able to write functions that rely on mutable global variables to produce their output clearly contradicts the stated property.

Proving correctness is not the only benefit of equational reasoning. Richard Bird, the author of the book titled *Pearls of Functional Algorithm Design* [65], shows how equational reasoning can be used to design efficient algorithms. Starting from an obviously correct but inefficient version of an algorithm, he iteratively rewrites it until reaching a optimised version.

### 3.2 Equational Reasoning of x86 Assembly Code

This section is entirely based on the paper titled *Equational Reasoning of x86 Assembly Code* [1] written by Kevin Coogan and Saumya Debray from the University of Arizona. The reason this work takes such a central place is that my contribution presented in Chapter 4 is based on their work.

The paper argues that there is a myriad of source code analysis tools focused on analysing correctness, efficiency and security of software application at source code level, but that there is a void for similar tools aimed at assembly code (either from disassembly or hand written sources). To overcome this problem, they developed a prototype which is able to perform dynamic analysis on assembly code for the Intel x86 architecture by means of equational reasoning. It works by first translating every instruction into a set of equations which encapsulates their exact semantic to form an equational reasoning system, and then manipulating the system in various ways to extract meaningful information.
Equational reasoning over assembly code is a novel application of equational reasoning. It has been chosen by the authors for it allows to accurately model the dependencies between instructions, which could be lost with other analysis tools. The dependencies arise from the many registers’ names and the implicit side effects most instructions have. Moreover, equational reasoning allows to improve the readability of the assembly code. These three topics are discussed in a more detailed manner below.

- **Register Name Aliasing**: As explained in Section 2.4.4.1, the four all purpose registers can be addressed in four different ways: As a whole, as the bottom 16 bits, and as the left half and the right half of the bottom 16 bits. This has been illustrated in Figure 3.1. The equational representation allows to modelise the relationship between a register and its many names to provide a more accurate analysis.

- **Side Effects**: Most instructions have side effects on the `eflags` register. For example, the add instruction will set the overflow flag to one if the arithmetic operation has overflowed. This register is then used to influence the instructions in charge of the conditional branching. When translating an instruction into a set of equations, a subset of the set will be dedicated to representing this behaviour.

- **Readability**: Because assembly languages are the lowest level of abstraction one could reach, they can be very verbose. A simple operation in a high level language will be translated into a set of many assembly instructions, making it harder to read, and so, to reason out. As a result, being able to visualise these instructions in a straightforward manner is of substantial help to reverse engineers. This can be arguably achieved by the use of an equational representation.

![Figure 3.1: $r = A, B, C, D$. Illustrates the dependencies between a register and its sub-parts.](image)

As for the choice of using the Intel x86 architecture, it comes from the fact that the authors intended to use their tool to analyse malware, which is usually written to target the most ubiquitous architecture. To be noted that one could easily extend their research by adding support for the x64 architecture, something that will not be discussed in this work.
3.2.1 Motivating Example

To illustrate what has been discussed so far, an example given by Kevin Coogan in his PHD thesis titled *Deobfuscation of Packed and Virtualization-Obfuscation Protected Binaries* [66] will be presented and broadly explained. The reason for this example to not be an original one is due to the fact that, at least to my knowledge, the tool made by the authors has not been released to the public. To be noted that the vocation of this example is not to exhaustively explain everything in detail, but rather to give a broad idea of why equational reasoning is helpful for performing reverse engineering.

The x86 assembly trace given in Listing 3.2 will be the input given to the tool. It performs operations on three registers, *eax*, *ebx*, and *ecx*. For this example, we are only interested in the value that *eax* will take once past the fifth instruction.

The Figure 3.2 shows the equations that have been generated from the trace. All left-hand side terms in the equation listing have a subscript which relates to the line numbers in the trace, and the right-hand side terms have subscripts relating to previous results of equations. The *const* subscript notifies that no previous information is known about an operand. One can see how the one-to-many mapping between instructions and equations allows to fully modelise the behaviour of each instruction. One might also have noticed that the very last instruction has been added manually. Because we are interested in the content of *eax* and not just *ax*, adding this equation allows the analysis to be performed in the whole register.

Finally, Figure 3.3 shows how the equational reasoning is applied. We start by saying that $eax_6 = eax_5$ and recursively substitute the terms by their definition found in Figure 3.2. Simplifications are performed whenever possible until reaching an irreducible expression, here $eax_6 = 0x1$.

```
0: xor ebx, ebx
1: not bx
2: mov eax, 0x7e5bd96f
3: mov ecx, 0x81a42692
4: and eax, ecx
5: add ax, bx
```

Listing 3.2: Snippet of assembly code. From the PHD thesis of Kevin Coogan [66].
Figure 3.2: Equational system generated from the trace found in Listing 3.2.

\[
\begin{align*}
ebx_0 &= ebx_{\text{const}} \oplus ebx_{\text{const}} \\
efl_0 &= \text{Flag}(ebx_{\text{const}} \oplus ebx_{\text{const}}) \\
ebx_1 &= \neg ebx_0 \\
bx_1 &= \text{Restrict}(ebx_1, 0011) \\
eax_2 &= 0x7e5bd96f \\
ecx_3 &= 0x81a42692 \\
eax_4 &= eax_2 \& ecx_3 \\
eax_4 &= \text{Restrict}(eax_4, 0011) \\
efl_4 &= \text{Flag}(eax_4 \& ecx_3) \\
ax_5 &= ax_4 + bx_1 \\
eax_5 &= (eax_4 \& 0xffff0000) | ax_5 \\
efl_5 &= \text{Flag}(ax_4 + bx_1) \\
eax_6 &= eax_5
\end{align*}
\]

Figure 3.3: Simplification of an equation about the \textit{eax} register. From the PHD thesis of Kevin Coogan [66].

3.2.2 Notation

As said in Section 2.4.4.2, the syntax of the Intel x86 ISA is as follows:

\textbf{label}: \textit{mnemonic argument1, argument2, argument3}

Most of the instructions have an arity less than or equal to 2 and use the first operand as a
source and destination operand. For example, the multiplication instruction `mul arg1, arg2` could be rewritten as `arg1 := arg1 * arg2`. An operand can be a register name, a constant (also called immediate value), or a memory location represented with an address expression. The expression is found enclosed in brackets and has to be evaluated before being used. For example, the instruction `mov eax, [ebx + 8]` will take the value at the address stored in `ebx` plus 8 and store it into `eax`.

Now it will be discussed the notation for the equations. Each instruction’s mnemonic will be mapped onto an operator that can be more easily understood whenever it is possible. Source operands can then be applied to the operator using either infix or prefix notation and it will give back a result called the destination operand. This is what has been done with the `mul` example from the previous paragraph. Just like with assembly instructions, an operand can either be a constant, a register name, or a memory expression. For the latest case, a memory expression will be represented by `MLOC[a..b]`, where `a..b` defines a memory range, and the value stored at the memory location will be represented by `ValueAt(MLOC[a..b])`.

Registers and memory locations will change their state over time, something that is not compatible with the ideas proposed by the authors so far. To get over this issue, every variable (either registers or memory location) is given an identifier to uniquely identify every state it has had. This will be done via the use of subscripts. The line number of each instruction in the trace will be referred as the order number and will be used as a source for unique identifiers.

A simple example showing the new notation can be observed below. In Listing 3.3 it is shown a trace prefixed with line numbers, in Listing 3.4 the generated equations. The `mov` is replaced by an equal sign, and the `add` by a plus sign. To be noted that, for the purpose of this example, the equations do not fully modelise the instructions.

| 0: mov eax, [401000] |
|---|---|
| 1: add eax, 2 |

Listing 3.3: A sample of trace.

\[
\begin{align*}
eax_0 & := ValueAt(MLOC[40100..401003])_{\text{const}} \\
eax_1 & := eax_0 + 2
\end{align*}
\]

Listing 3.4: A partial translation of the trace from Listing 3.3.
3.2.3 Implementation

3.2.3.1 Translating Instructions

Each instruction has to be converted into a set of equations which fully modelise the behaviour of the instruction. To do so, the tool will linearly pass over the trace and perform the translation. The order number of each instruction will be used as a unique identifier for the destination operands (left-hand side) of its generated set of equations. As for the source operands, since nothing is known about them yet, the bottom (⊥) symbol will be used instead. This will be replaced by valid identifiers later on when dependencies are being resolved.

The push and div instructions will be used as examples to illustrate the conversion. In the situation where push eax is seen as a complete trace, the following set of equations will be generated:

\[
\begin{align*}
\text{ValueAt}(MLOC[1000..1003])_0 & := \text{eax} \perp \\
\text{esp}_0 & := \text{esp}_\perp - 4
\end{align*}
\]

The stack is simply a special usage of the memory combined with dedicated registers. Putting the value manually on top of the stack (at address 1000 in this example) and updating the stack pointer register to the newest top position is equivalent to using push. If div eax, 2 were seen as a complete trace, the following equations would be generated:

\[
\begin{align*}
\text{eax}_0 & := \text{eax}_\perp / 2 \\
\text{eflags}_0 & := \text{Flag}(\text{eax}_\perp / 2)
\end{align*}
\]

Here, the eflags register has to be updated because 6 flags can possibly be changed. This is done thanks to the new equation Flag, which takes the expression as the only source operand.

3.2.3.2 Resolving Dependencies

Here it will be discussed how the bottom symbols are replaced by identifiers to resolve the dependencies between the equations. This process is not straight forward for two reasons, the first one being that registers can be accessed using different names to read and write different parts of them, and the second one from the fact that the Intel x86 architecture is byte addressable.
Resolving dependencies will be done by going backward through the listing of equations and looking where the source operands have been declared. There are five scenarios that the algorithm must handle to correctly resolve the dependencies. They will be described in the list below, and an example for each scenario can be found in Table 3.1.

1. **First case**: A source operand is fully defined by a previous destination operand. In this case, the source operand will simply take the identifier of the destination operand.

2. **Second case**: A source operand is a subset of another destination operand. For registers, it could be $ch$, which is a subset of $ecx$, and for memory location, it could be $MLOC[1000..1001]$, which is a subset of $MLOC[1000.10003]$. Because they are not equal, it is required to refine the most general operand to match the other one. This is done with the *Restrict* equation, which takes 2 operands, a register or memory location, and a mask to tell which part to isolate. When scanning backward for the definition of an operand, the tool will then have to detect when this case applies and add the *Restrict* equation with a correct mask.

3. **Third case**: It is the opposite of the second case. A source operand is defined by multiple previous destination operands. For example, $ah$ and $al$, which defines $ax$. To handle this situation, the tool has to detect the parts that form the whole and add equations to each of them to progressively recompose the whole. This can be observed in line 3 and 4 from the example.

4. **Fourth case**: A source operand is made from parts of multiple destination operands while none is a subset of the other. This is impossible for registers but not for memory locations. To deal with this case, it is necessary to combine the solutions of the second and third case.

5. **Fifth case**: When a source operand cannot be traced to a destination operand, nothing can be said about it. In this case, the identifier will be *const*. This can happen at the beginning of the program when the registers have not been initialised yet and also when dealing with obfuscated code.

### 3.2.3.3 Applying Equational Reasoning

Once every instruction has been translated and the dependencies have been resolved, it is possible to reason about the equational system. To analyse what a variable has been through at a specific location in the trace, one has to first insert a new equation of the form $var_{line\ number} := var_{\perp}$ and then let the tool substitute operands by their definition. The equation that is progressively formed by the successive substitutions could also be simplified using rewriting rules. This is what has been shown in Figure 3.3.
<table>
<thead>
<tr>
<th>Case</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
</table>
| 1:   | $eax_0 := 40$  
$eax_1 := eax_1 + 2$ | $eax_0 := 40$  
$eax_1 := eax_0 + 2$ |
| 2:   | $eax_0 := FFFh$  
$ah_1 := ah_0 \oplus bh_1$ | $eax_0 := FFFh$  
$ah_0 := \text{Restrict}(eax_0, 0010)$  
$ah_1 := ah_{ah_0} \oplus bh_{cons}$ |
| 3:   | $eax_0 := 4000$  
$ah_1 := 10$  
$al_2 := 10$  
$eax_3 := eax_1 + 2$ | $eax_0 := 4000$  
$ah_1 := 10$  
$eax_1 := (eax_0 \& 0010) | ah_1 < 6$  
$al_2 := 10$  
$eax_2 := (eax_1 \& 0001) | al_2$  
$eax_3 := eax_2 + 2$ |
| 4:   | $\text{ValueAt}(MLOC[0..3])_0 := FFFh$  
$\text{ValueAt}(MLOC[4..7])_1 := FFFh$  
$eax_2 := \text{ValueAt}(MLOC[2..5])_1$ | $\text{ValueAt}(MLOC[0..3])_0 := FFFh$  
$\text{ValueAt}(MLOC[2..3])_0 := \text{Restrict}(\text{ValueAt}(MLOC[0..3])_0, 0011)$  
$\text{ValueAt}(MLOC[4..7])_1 := FFFh$  
$\text{ValueAt}(MLOC[4..5])_1 := \text{Restrict}(\text{ValueAt}(MLOC[4..7])_1, 1100)$  
$\text{ValueAt}(MLOC[2..5])_1 := (\text{ValueAt}(MLOC[2..3])_0 << 16)$  
$\text{ValueAt}(MLOC[4..5])_1$  
$eax_2 := \text{ValueAt}(MLOC[2..5])_1$ |
| 5:   | $eax_0 := ebx_1 + ecx_1$ | $eax_0 := ebx_{\text{const}} + ecx_{\text{const}}$ |

Table 3.1: Examples for the 5 situations one can encounter when resolving dependencies.
Chapter 4

Contribution

4.1 Idea and Motivation

In the previous chapter, the work of Kevin Coogan and Saumya Debray, presented in the paper titled *Equational Reasoning of x86 Assembly Code* [1], has been discussed. In the frame of their work, they needed a tool to assist them with analysing traces of Intel x86 instructions from malware. They then put forward a set of rules to translate instructions into equations, and a term-rewriting system to manipulate them. Even if the tool has not been made public, one could write its own version of it for the pseudo code has been published in Kevin Coogan’s PHD thesis [66].

The contribution of this work will be to show how to extend their tool by allowing it to work in a static analysis context. The authors broadly discussed this idea in the penultimate section, but without diving in depth in the topic. They proposed to turn the assembly code into an intermediate language of type Static Single Assignment, or SSA for short, and they pointed toward a paper to solve the problem of aliasing which comes with indirect memory accesses. In the following sections of this chapter, it will be given a description of the procedures that are required to make their ideas practical for static analysis.

4.2 Complications

4.2.1 Branching

In the dynamic analysis case, the various definitions a variable can have are uniquely differentiated by the order number of the defining instruction. Because traces contain the sequence of instructions that have been executed by the CPU in a sequential order, there will not be any branching, or in another words, there will not be the possibility to go back up in the trace. As a result, it is enough to use the order number as a source for unique identifiers.
This is in contrast with the static analysis case, where listings of instructions can contain conditional and unconditional branching instructions. The problematic situation appears when branching toward a part of the listing that has already been seen. It would be necessary to redefine some of the definitions that are used as source operands in between the landing point and the branching instruction to account for the changes of state that have happened during the previous pass. This is in contradiction with the non mutating state property that is essential to the equational reasoning.

In Listing 4.1, this phenomenon can be observed. To be noted that, for the sake of the example, the two notations have been mixed and the code itself does not serve any purpose. For the new state of eax to be reflected when going back up, one would have to add a definition in the form of $eax_{30} := eax_{31}$ between the equations at line 31 and 32.

\[ 30 : eax_{30} := ebx_{29} + eax_{28} \]
\[ 31 : eax_{31} := eax_{30} - 1 \]
\[ 32 : jz 30 \]

Listing 4.1: Example where the branching instruction disrupt the analysis.

In a static analysis situation, one can actually only use the equational reasoning tool of Kevin Coogan and Saumya Debray on basic blocks. Since they are sequences of instructions with no branches getting in except for the entry point, and no branches going out except for the last instruction, the definitions will not have to mutate.

### 4.2.2 Indirect Memory Access

The other issue, which is inherently linked with static analysis, is having indirect memory accesses, that is, having one of the operands of an instruction being a memory expression that has to be computed first. The Intel x86 instructions able to access memory locations are given operands that point to these locations by means of the addressing mode shown in Equation 4.1. The brackets indicates optional parameters, at least one of the three brackets has to be used. When dealing with traces, the values of the registers are well known and so, it is easy to find out which memory location is being accessed and or modified. On the contrary, in a static context, it is not always possible to gain this knowledge for registers’ values could only be known at runtime.
The equational reasoning relies on the fact that the dependencies between definitions and usages can be traced by simply going upward in the trace. When encountering an operand which is a memory location that has to be dynamically computed, it might not be possible to correctly resolve further dependencies because of unknown aliasing relationships with that operand.

An example can be observed in Listing 4.2. In this scenario, the registers ebp and esp are pointing toward the same memory location. Both registers are marked as constant for the sake of the example. The first and second lines are setting the same location in memory to a different value, the third line is using the value stored in the memory location to perform an exclusive or. Because the algorithm which performed the translation into the equational form did not realised \([ebp]\) and \([esp]\) were aliased, it uses the old value, 42, as one of the two operands of the exclusive or. As a result, the translation algorithm produced an erroneous listing.

\[
\begin{align*}
V_{\text{al}}(\text{MLOC}[esp_{cons}..esp_{cons} + 3])_1 & := 42 \\
V_{\text{al}}(\text{MLOC}[ebp_{cons}..ebp_{cons} + 3])_2 & := 43 \\
eax_3 & := eax_{cons} \oplus V_{\text{al}}(\text{MLOC}[esp_{cons}..esp_{cons} + 3])_1
\end{align*}
\]

Listing 4.2: Example of a problematic situation that arose from indirect memory accesses.

4.3 Static Single Assignment

Static Single Assignment is a property a language can have. It states that each variable can only be assigned once. As a consequence, languages with this property are referentially transparent. Indeed, if a variable cannot be reassigned, states cannot mutate. Andrew W. Appel even argues in his paper titled SSA is Functional Programming [68], that, without too many surprises, SSA is indeed functional programming.
One might have noticed that the language presented in the previous chapter is in SSA form. Each variable definition is assigned a unique identifier which is the line where the original instruction appears in the trace, and each uses of a variable is renamed to match the definition’s new name.

Languages based on SSA are widely used by compilers to perform optimisations such as constant propagation [2], code motion [3], and elimination of partial redundancies [4] because it is a very efficient way of representing the data flow of programs. They operate as follows: First a source code is turned into an SSA form, then the SSA form is applied to as many optimisation algorithms as possible, and finally the SSA form is translated back into either the source language or another language.

The SSA form has been introduced by Ron Cytron et al in the paper titled *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph* [5] published in 1991. The efficient algorithm they proposed to turn a program into an SSA form requires a control flow graph as input. In a reverse engineering context, this is just what one would want for most tools provide this representation by default.

Contrary to the representation of Kevin Coogan and Saumya Debray, the SSA form provides a way to follow the dependencies when dealing with branching. It is done thanks to the $\phi$-function, which is a special kind of assignment that takes two or more definitions of the same variable and turns them into a new definition. This can be observed in Listing 4.4. Because of the *do/while* construct, the $x := x \ast 2$ statement can be executed more than once, and so requires special care. The semantic of the $\phi$-function is that, if the control flow comes from the first assignment, $x_2$ will be equal to $x_1$, and if the control flow comes from the loop construct, $x_2$ will be equal to $x_3$.

```
x := 1
do
    x := x \ast 2
while P
```

Listing 4.3: A simple while loop

```
x_1 := 1
do
    x_2 := \phi(x_1, x_3)
x_3 := x_2 \ast 2
while P
```

Listing 4.4: Translating a while loop found in Listing 4.3 into a SSA form using a $\phi$ function.

The remaining parts of this section will be about defining more formally the notion of control flow graph, and then giving the procedure to translate code into a SSA form.
4.3.1 Control Flow Graph

A CFG is a directed graph whose nodes are basic blocks and where edges represent the transfer of control between these blocks. To be complete, two more nodes are added: The entry node that connects every basic block from which the program can be entered, and the exit node which is connected to every block that can exit the program. In this configuration, every node is on at least one path from Entry and one path to Exit. Each variable used in any of the basic block has been initialised in the Entry block to whatever value which may represent the starting state of these variables.

An edge from the block $X$ to the block $Y$ will be represented by $X \rightarrow Y$. The successors of a node $X$ are every node $Y$ with an edge $X \rightarrow Y$. The predecessors of a node $X$ are every node $Z$ with an edge $Z \rightarrow X$. The set of all successors of a node $X$ will be then represented by $Succ(X)$, and the set of all predecessors by $Pred(X)$. A joint node is a node that has more than one predecessor.

A non-null path from node $X_0$ to node $X_j$ of size $J$ will be denoted as $X_0 \xrightarrow{\_} X_j$. Two non-null paths $X_0 \xrightarrow{\_} X_j$ and $Y_0 \xrightarrow{\_} Y_k$ converge at node $Z$ if:

$$ X_0 \neq Y_0 $$
$$ X_j = Z = Y_k $$
$$ (X_j = Y_k) \implies (j = J \text{ or } k = K) $$

Intuitively, two non-null paths converge if they join at the end.

4.3.2 Translating into SSA form

According to Cytron et al [5], a program is in an SSA form if it meets these three conditions:

- **First condition**: Two non-null paths $X \xrightarrow{\_} Z$ and $Y \xrightarrow{\_} Z$ converge at a node $Z$, and nodes $X$ and $Y$ contain assignment to $V$ in the original program, then a trivial $\phi$-function $V \leftarrow \phi(V, ..., V)$ has been inserted at $Z$ in the new program.

- **Second condition**: Each mention of $V$ in the original program or in an inserted $\phi$-function has been replaced by a mention of a new variable $V_j$, leaving the new program in SSA form.

- **Third condition**: Along any control flow path, consider any use of a variable $V$ in the original program and the corresponding use of $V_i$ in the new program. Then $V$ and $V_i$ have the same value.
Translating a program into an SSA form is then done in two steps:

- **First step**: It consists of inserting trivial $\phi$-functions at the entrance of certain join nodes in the CFG. They will have the following form: $V \leftarrow \phi(W,X,..)$. The amount of operands applied to the $\phi$-function will depend on how many predecessors the node has. The predecessors will be listed in a fixed order, and the $j$th operand of $\phi$ will be associated with the $j$th predecessor. This simply means that, if the control flow comes from the $j$th node, the $j$th operand will be selected by the $\phi$-function.

- **Second step**: It consists of replacing each mention of a variable $V$ by a new variable $V_j$ so that the three properties stated above hold.

An SSA form which has a minimal amount of $\phi$-functions, while keeping the first condition true, is said to be in minimal SSA form. Another flavour of SSA is called pruned SSA [6] form, and has the particularity of not having $\phi$ function for variable that are not live in the rest of the program. In our situation, we want the minimal form so that we can analyse the data flow of a variable at any point in the code. The steps to turn a program into a minimal SSA form will then be described in the following sections.

4.3.2.1 **Setting the $\phi$-functions**

A naïve approach to finding out where to put the $\phi$-functions would be to enumerate every pairs of assignment for the same variable and verify if they can reach a common node. The problem with this method is that it is not something that can be achieved in linear time. Another way to find out where to make the insertions is to use the dominance frontier of every node.

Before explaining what is a dominance frontier, it is first necessary to lay down a little bit of terminology.

- For two nodes $X$ and $Y$ from a CFG, it is said that $X$ dominates $Y$ if $X$ is on every path from the Entry node to $Y$. This relationship will be denoted as follows: $X \geq Y$.

- If $X$ dominates $Y$ and $X \neq Y$, it is said that $X$ strictly dominates $Y$. This relationship will be denoted as follows: $X > Y$.

- If $X$ does not strictly dominate $Y$, the following notation will be used: $X \nexists Y$.

- The immediate dominators of a node $X$ are the closest strict dominators of $X$ on the paths from Entry to $X$ on the CFG. A node can have more than one immediate dominators. The set of all immediate dominators of a node $X$ will be denoted as follows: $idom(X)$. 
• $\text{Dom}(X)$ represents the set of all nodes that dominate $X$.

The dominance frontier of a node $X$ is the set of all nodes $Y$ that are not strictly dominated by $X$ while having at least one successor which is dominated by $X$. More formally, the dominance frontier can be defined as:

$$DF(X) = \{ Y | \exists P \in \text{Pred}(Y), \, X \geq P \text{ and } X \not\prec Y \}$$

To better illustrate the concept of dominance frontier, let’s use the CFG found in Figure 4.1 as an example. Each node is identified by its number, and, as said before, the entry node initialises every variable to some value to represent the state of the program at its start. Node 4 dominates nodes 5, 6, and 7. The dominance frontier of node 4 is then nodes 3, 10, and 9.

![Control Flow Graph](image)

**Figure 4.1: Example of control flow graph which requires $\phi$-functions.**

In the context of the same example, let’s say variable $V$ gets redefined in node 4. Node 5, 6, and 7 will not need $\phi$-functions for that variable because they will only be exposed to the definition of node 4. Node 9, on the other hand, will be exposed to either the definition of the entry node or the definition of node 4. It then requires a $\phi$-function.

The algorithm used to find out the dominance frontier of every node in the CFG is given in Alg 1. As an input, it takes a CFG, but also a dominator tree. The dominator
tree is a data structure where each node has for children the nodes it immediately dominates, and where the root node is the entry node. The dominator tree can be computer in linear time with the algorithm presented by Thomas Lengauer and Robert Tarjan in an almost linear time [7]. In the algorithm, \( \text{Children}(X) \) relates to the children of a node in the dominator tree.

**Algorithm 1** Algorithm proposed by Cytron et al [5] to compute the dominator frontier of each node of a CFG.

\[
\begin{align*}
1: & \text{ for each } X \text{ in a bottom up traversal of the dominator tree do} \\
2: & \quad DF(X) \leftarrow \emptyset \\
3: & \quad \text{ for each } Y \in \text{Succ}(X) \text{ do} \\
4: & \quad \quad \text{ if } \text{idom}(Y) \neq X \text{ then} \\
5: & \quad \quad \quad DF(X) \leftarrow DF(X) \cup \{Y\} \\
6: & \quad \quad \text{ end if} \\
7: & \quad \text{ end for} \\
8: & \text{ for each } Z \in \text{Children}(X) \text{ do} \\
9: & \quad \text{ for each } Y \in DF(Z) \text{ do} \\
10: & \quad \quad \text{ if } \text{idom}(Y) \neq X \text{ then} \\
11: & \quad \quad \quad DF(X) \leftarrow DF(X) \cup \{Y\} \\
12: & \quad \quad \text{ end if} \\
13: & \quad \text{ end for} \\
14: & \text{ end for} \\
\end{align*}
\]

And finally, the algorithm to place the \( \phi \)-functions is given in Alg 2. \( \text{Work}(\ast) \) and \( \text{HasAlready}(\ast) \) are arrays of flags, and \( A(V) \) is the set of nodes which contain an assignment to \( V \). For the proof of correctness of these two algorithms, as well as their complexity analysis, one should refer to the authors’ paper [5].

4.3.2.2 Variable Renaming

First of all, it is necessary to give the form the assignments will obtain in the SSA form. An assignment \( A \) will be turned into \( \text{LHS}(A) \leftarrow \text{RHS}(A) \), where \( \text{LHS}(A) \) is a tuple of distinct variables, and where \( \text{RHS}(A) \) is a tuple of expressions. Obviously, it is required for these two tuples to be of equal size for the variables get assigned to the value of the expressions. The use of tuple is necessary for constructs such as function calls, where multiple variables can be defined at the same time.

It is also required to describe the data structure which will be used by the algorithm: \( \text{C}(\ast) \) is an array of integers which records how many assignments every variable has been exposed to so far, \( \text{S}(\ast) \) is an array of stacks, one per variable, which contains integers,
Algorithm 2 Algorithm proposed by Cytron et al [5] to insert the $\phi$-functions.

1: $IterCount \leftarrow 0$
2: for each node $X$ do do
3: $HasAlready \leftarrow 0$
4: $Work \leftarrow 0$
5: end for
6: $W \leftarrow \emptyset$
7: for each variable $V$ do
8: $IterCount \leftarrow IterCount + 1$
9: for each $x \in A(V)$ do
10: $Work(X) \leftarrow IterCount$
11: $W \leftarrow W \cup \{X\}$
12: end for
13: while $W \neq \emptyset$ do
14: Take $X$ from $W$
15: for each $Y \in DF(X)$ do
16: if $HasAlready(Y) < IterCount$ then
17: place $V \leftarrow \langle \phi(V, ..V) \rangle$ at $Y$
18: $HasAlready(Y) \leftarrow IterCount$
19: if $Work(Y) < IterCount$ then
20: $Work(Y) \leftarrow IterCount$
21: $W \leftarrow W \cup \{y\}$
22: end if
23: end if
24: end for
25: end while
26: end for
and where the top of the stacks contains the value $i$ used to construct the variables, $WhichPred(X, y)$ is an integer that represents which predecessor of $Y$ in the CFG $X$ is, and $oldLHS(a)$ is the original tuple.

The algorithm for renaming can be observed in Alg 3. It begins by initialising $C(\ast)$ and $S(\ast)$, and then it starts a top down traversal on the dominator tree, beginning with the entry node. This can be observed between line 1 and line 5. The search function will handle the renaming. Its first loop will do the renaming of the $RHS$ variables which are not part of a $\phi$-function, and every $LHS$ variable. This can be observed between line 8 and line 20. The next loop, between line 21 and line 26, will handle the renaming of the $RHS$ variables that are part of a $\phi$-function. The recursive descent is handled between line 27 and line 29. Finally, some bookkeeping is done between line 30 and line 34.

4.3.3 Memory Aliasing

"As a result of alias issues, memory expressions must be divided into those which are safe to propagate, and those which must not be propagated at all."

Michael James Van Emmerik, 2007

As explained before in Section 4.2.2, the indirect memory accesses cause problems in a static context for it is not easy to follow the data flow, and as a result, propagating values gets tricky. This is due to the fact that there is more than one way to refer to the same memory location. The causes of these aliasings are mostly from the manipulation of the stack, and the frame pointers [8]. Fortunately, there cannot be such aliasing problems with registers. Indeed, the only way to change the content of, or to refer to content of, let’s say, eax is to explicitly specify eax, or one of its three other names, as an operand.

One of the many solutions would be the following: Heap storage can be modelised as one single variable that is redefined every time one of its region is updated. This approach is not very conservative but still allows optimization to be done [5]. Unfortunately, in our context, we do not only want to apply optimisation, but also to provide a way to follow the data flow in certain regions of the program.

Another solution would be to not propagate $LHS$ variables that are defined by functions which are applied to at least one memory expression, but this is not what one would want to have. Propagation is possible, easily inside basic blocks, and in a more complicated way across basic blocks.
Algorithm 3 Algorithm proposed by Cytron et al [5] to rename the variables.

1: for each variable $V$ do
2: \hspace{1em} $C(V) \leftarrow 0$
3: \hspace{1em} $S(V) \leftarrow \text{EmptyStack}$
4: end for
5: call search(Entry)

6: function search($X$)
7: for each statement $A$ in $X$ do
8: \hspace{1em} if $A$ is an ordinary assignment then
9: \hspace{2em} for each variable $V$ used in $RHS(A)$ do
10: \hspace{3em} replace use of $V$ by use of $V_i$, where $i = \text{Top}(S(V))$
11: \hspace{2em} end for
12: \hspace{1em} end if
13: \hspace{1em} for each $V$ in $LHS(A)$ do
14: \hspace{2em} $i \leftarrow C(V)$
15: \hspace{2em} replace $V$ by new $V_i$ in $LHS(A)$
16: \hspace{2em} push $i$ onto $S(V)$
17: \hspace{2em} $C(V) \leftarrow i + 1$
18: \hspace{1em} end for
19: end for
20: for each $Y \in \text{Succ}(X)$ do
21: \hspace{1em} $j \leftarrow \text{WhichPred}(Y, X)$
22: \hspace{1em} for each $\phi$-function $F$ in $Y$ do
23: \hspace{2em} replace the $j$-th operand $V$ in $RHS(F)$ by $V_i$ where $i = \text{Top}(S(V))$
24: \hspace{2em} end for
25: end for
26: for each $Y \in \text{Children}(X)$ do
27: \hspace{1em} call search($Y$)
28: end for
29: for each assignment $A$ in $X$ do
30: \hspace{1em} for each $V$ in $oldLHS(A)$ do
31: \hspace{2em} $\text{pop}S(V)$
32: \hspace{2em} end for
33: \hspace{1em} end for
34: end for
35: end function
As said before, one could reason about memory locations from within a basic block. Two memory locations $i$ and $j$ are non-conflicting if at least one of the two following conditions hold:

- The memory location $i$ uses a register known to point to the stack, while the memory location $j$ points to the heap.
- They use the same base register but different offset, and the base register is not redefined in between the two memory locations.

This is again not enough for we want something which allows data to be followed outside basic blocks.

A third solution, one that would be satisfactory, is the one proposed by Gogul Balakrishnan et al in the paper titled *Analyzing Memory Accesses in x86 Executables* [9]. In it, they describe a static analysis algorithm for x86 executable files called the value-set analysis, which yields an over-approximation of the set of values each data object can hold at each program point. A data object can either be a memory location or a register. To modelise the data objects, they introduced the concept of abstract locations, or a-locs for short. Intuitively, an a-loc can be roughly compared to a variable in a programming language such as $C$. More precisely, a-locs are based on the fact that generating an executable from a high level language comes after establishing the data layout of the program: Global variables will be accessed through static addresses, and local variables will be accessed through static stack frame offsets that are added or subtracted to either $esp$ or $ebp$. An a-loc is simply a set of locations between two statically known locations/offsets. To be noted that they thus cannot overlap.

To illustrate the results the value-set analysis would give, let’s use the example provided in the paper [9]. It is not an original one for the tool performing the analysis is not freely available. The original C code can be found in Listing 4.5, its assembly version can be found in Listing 4.6. To be noted that the C code is not used by the value-set analysis and has only been added to make the example easier to understand from the reader’s perspective. The purpose of the code is to fill the first half of the $a$ array with 0s, to fill the other half with 1s, and finally to return the first value of $a$.

Upon inspection of the assembly code, one would notice that variables $part1$, $part2$, and $i$ have been replaced by registers, respectively $eax$, $ebx$, and $ecx$. Also, the two global variables, $part1Value$ and $part2Value$, are stored at addresses 4 and 8, respectively.
```c
int part1Value = 0;
int part2Value = 1;

int main() {
    int *part1, *part2;
    int a[10], *p_array0;
    int i;

    part1=&a[0];
p_array0=part1;
part2=&a[5];

    for (i=0; i<5; i++) {
        *part1=part1Value;
        *part2=part2Value;
        part1++;
        part2++;}

    return *p_array0;
}
```

Listing 4.5: Sample of C code used for the value-set analysis. This has been taken from the paper of Gogul Balakrishnan et al [9].

```assembly
proc main
1: sub esp , 44
2: lea eax , [esp+4]
3: lea ebx , [esp+24]
4: mov [esp+0], eax
5: mov ecx , 0
6: mov edx , [4]
7: mov [eax], edx
8: mov edx, [8]
9: mov [ebx], edx
10: add eax , 4
11: add ebx , 4
12: inc ecx
13: cmp ecx , 5
14: jl 6
15: mov edi , [esp+0]
16: mov eax , [edi]
17: add esp , 44
18: retn
```

Listing 4.6: Assembly code resulting from the C code found in Listing 4.5. This has been taken from the paper of Gogul Balakrishnan et al [9].

On the left side of Figure 4.2, it can be observed the data layout of the compiled program from Listing 4.5. The stack frame contains the local variables, being the array of 10 integers `a`, and `p_array0`. The two global variables are somewhere else, outside the stack. On the right, one can see the two memory regions that the value-set analysis would detect. Memory regions are continuous parts of the memory space of the program. There is one per memory allocation statement (malloc), one for the global region, and one for each procedure. To be noted that the AR in AR-main stands for activation record, which is another name given to a stack frame. For the value-set analysis, memory addresses are made by a pair memory region-offset. For example, `part1Value` is located at address \((\text{Global}, 4)\).

The a-locs of the example can also be observed on the right side of Figure 4.2. The a-loc \(\text{var}_{40}\) represents the set of locations between \(\text{var}_{20}\) and \(\text{var}_{44}\), that is, \(a[0]\) to \(a[4]\) included. The a-loc \(\text{var}_{44}\) represents the set of locations between \(\text{var}_{44}\) and the
end of the AR-main region, that is, the end of the stack frame. It maps to $p.array0$ in the data layout.

The algorithm over approximates the set of values each a-loc can take, and to represent the over approximation, it uses the notion of reduced interval congruence, or RIC for short. A RIC can be represented as a tuple of 4 elements, $(a,b,c,d)$, and it means $a \ast [b,c] + d$. Formally, it denotes the set $\{aZ + D | Z \in [b,c]\}$. As an example, $(2, 0, 4, 1)$, or $2 \ast [0, 4] + 1$, represents the set $\{1, 3, 5, 7, 9\}$.

Finally, the value-set analysis would yield the following result for the entry of main: ${\{esp \rightarrow (\bot, 0), \text{mem.4} \rightarrow (0, \bot), \text{mem.8} \rightarrow (1, \bot)\}}$. The first element of each tuple that is pointed to by the a-locs corresponds to the global memory region, and the second element corresponds to the AR-main memory region. The results tell us that $esp$ will not have any meaningful values in the global region, and that it will have the value 0 in the AR-main region. If the analysis was to be run on line 7, it would yield the following results:

$${\{esp \rightarrow (\bot, -44), \text{mem.4} \rightarrow (0, \bot), \text{mem.8} \rightarrow (1, \bot), eax \rightarrow (\bot, 4[0, \infty] - 40),
\text{ebx} \rightarrow (\bot, 4[0, \infty] - 20), \text{var.44} \rightarrow (\bot, -40), ecx \rightarrow ([0, 4], \bot)\}} \tag{4.2}$$
If it was run for line 16, this would have been the results:

\[
\{ 
\begin{array}{l}
\text{esp} \rightarrow (\bot, -44), \text{mem}_4 \rightarrow (0, \bot), \text{mem}_8 \rightarrow (1, \bot), \text{eax} \rightarrow (\bot, 4[1, \infty] - 40), \\
\text{ebx} \rightarrow (\bot, 4[1, \infty] - 20), \text{var}_{44} \rightarrow (\bot, -40), \text{ecx} \rightarrow ([5, 5], \bot), \text{edi} \rightarrow (\bot, -40)
\end{array}
\}
\] (4.3)

Because the analysis determined that `edi` can take values from the set \(\{0, 1, 2, 3, 4\}\), it is clear that \([\text{eax}]\) and \([\text{ebx}]\) are not aliased. Reminder that \([\text{eax}]\) is *part1*, and that \([\text{ebx}]\) is *part2*.

For more information about the value-set analysis, one should refer itself to the paper [9]. The analysis proposed in it has been implemented in the form of a non-free plug-in for the IDA Pro framework called *CodeSurfer/x86*1. At the time of writing, and to the best of my knowledge, it is the only implementation of the value-set analysis.

### 4.4 Implementation and Difficulties

In this section, it will be described how one could implement the tool proposed by Kevin Coogan and Saumya Debray, which has been presented in Chapter 3, to work in a static context. As a reminder, their tool is only able to process traces of x86 assembly code, and so only provides dynamic analysis. Afterwards, the limitations that will suffer the resulting tool will be discussed.

#### 4.4.1 Possible Implementation and Difficulties

The tool will take as input an executable file, and it will yield a control flow graph where nodes will be containing code in an SSA form. Moreover, the tool will have to allow analysis of registers and memory locations, as the tool of Kevin Coogan and Saumya Debray can do. To implement it, one can either start from scratch, or build up on top of an existing foundation. The later option will be described.

IDA Pro is a reverse engineering framework which has a great reputation in the reverse engineering world. It provides a disassembler, a debugger, but it also provides a scripting engine and a software development kit which allows programmers to extend the capabilities of the tool [10]. These capabilities and the fact that CodeSurfer/x86 is also a plug-in of IDA Pro, motivated the choice of IDA for the potential implementation of the tool.

CodeSurfer/x86 is a plug-in for IDA Pro which makes use of the value-set analysis explained in Section 4.3.3. It is used to generate intermediate representations for x86 programs, which can subsequently be explored through a graphical interface, or through a programming API and its scripting language. Amongst all the things it provides

---

figures a pointer analysis, which allows to see which pointers point to which variables. For more information about its other capabilities, one should refer itself to the official website\(^2\).

The architecture of CodeSurfer/x86, and how it interacts with IDA Pro, can be observed in Figure 4.3. The connector will first create data structures necessary to CodeSurfer/x86 by using the information coming from IDA Pro. The connector then performs the value-set analysis, and it passes information along to CodeSurfer. From CodeSurfer, a programmer can obtain the results of the pointer analysis as well as the control flow graph of the program being analysed using the API, which is accessible both in Scheme and C. The many other functionalities of CodeSurfer are not of interest for this work, and so they will not be discussed.

Finally, if one wants to implement the tool, he or she will have to apply the following operations:

1. Get the CFG from IDA Pro, or the one from CodeSurfer.
2. Get the points-to sets from CodeSurfer.
3. Go through the points-to sets to clearly identify aliased memory expressions.
4. Apply the translation from instruction to equations which has been explained in Section 3.2.3.1, without putting subscripts. Depending on the result of step 3, additional equations will have to be added to handle aliasing.
5. Apply the algorithm of Cytron et al [5] to insert the $\phi$-functions. The algorithm can be observed in Alg 2.

---

\(^2\)See https://www.grammatech.com/products/codesurfer again.
6. Apply the second algorithm of Cytron et al [5] to perform the renaming using subscripts. The algorithm can be observed in Alg 3.

7. Build up the equations for chosen registers and memory locations by simply looking for the latest reaching definitions. This has been explained in Section 3.2.3.3.

The fourth step mentions that additional equations will have to be added to handle aliasing. Fred Chow et al proposed a way to represent these aliasing relationships in the paper titled *Effective Representation of Aliases and Indirect Memory Operations in SSA Form* [12] by means of, amongst other things, *MayDef* definitions. This special definition takes as single operand the variable that may be modified, and it gives back another definition of the same variable. It is stated that a MayDef definition only potentially redefine a variable, and so leaves the possibility for the previous definition of that variable to be still referenced. It is modelised by the $\chi$-function, and it could looks like this: $v_x := \chi(v_{x-1})$.

In our case, we would like to also see what is causing the possible redefinition, and so, it would be preferable to add its possible value (or the location containing the value) as a second operand. This can be observed in Listing 4.8. To be noted that *ValueAt* has been replaced by brackets for readability reasons. In Listing 4.7, we know, thanks to previous analysis, that they are referencing to the same exact location for all execution path, and so it is possible to use a more straight forward approach.

![Listing 4.7](image1)

Listing 4.7: The memory location a and b are referencing to the same location.

![Listing 4.8](image2)

Listing 4.8: The memory location a and b may be referencing to the same location.

The difficulties in implementing this tool reside in the fact that IDA Pro and CodeSurfer/x86 are not easily accessible for they need to be purchased, but also because of the amount of work that implementing the algorithm of Kevin Coogan requires. The Intel x86 architecture posses many hundreds of instructions, which should be handled by the translation part of the algorithm. One short-cut would be to only implement the translation for the most common mnemonics. Peter Kankowski has disassembled three popular open-source applications which were compiled with the Microsoft Visual C++ 6.0 compiler, and he displayed the frequency of apparition of each mnemonic in a pie chart. The pie
chart can be observed in Figure 4.4, and his analysis can be found in the strchr blog\(^3\).

Figure 4.4: Frequency of mnemonic in the Intel syntax. Chart made on https://www.meta-chart.com/.

4.4.2 Limitations

The limitations come from the additions required to make the tool work in a static context. Two problems had to be resolved: Branching, and memory aliasing due to indirect memory accesses. They both come at a price that will be discussed hereunder.

4.4.2.1 SSA Form

The SSA form makes use of the $\phi$-functions to handle the redefinitions. These functions will arguably deteriorate the readability of the code, which was one of the arguments used by the authors of the original tool to justify its creation. The example found in Listing 4.9 will be used to illustrate the point. The code is made of three basic blocks, line 1 and 2 being the first one; line 3, 4, and 5 being the second one; and finally, line 6 being the third one. In this example, a $\phi$-function has to be inserted for eax, as seen in the CFG of the SSA form of the code in Figure 4.5. To be noted that the CFG does not include all the equations which interact with eflags.
Listing 4.9: Simple assembly code.

```
1: mov eax, 0
2: mov ebx, 5
3: add eax, 1
4: cmp eax, 5
5: jz 3
6: add eax, ebx
```

Figure 4.5: Control flow graph of the SSA form of the code found in Listing 4.9.

One can notice that the subscripts will not necessarily correspond to the line number of the instructions. For example, line 3 in Listing 4.9 does not correspond to the equation with a subscript of 3 in the CFG of Figure 4.5. Also, when substituting the operands with their definition, the $\phi$-function will get in the way. For example, starting with $eax_6 := eax_4 + ebx_2$, one can obtain the following result: $eax_6 := \phi(0, eax_4) + 1 + 5$. This equation does not reflect on the recursive aspect of the code. It would maybe be necessary to show it next to the same equation, where the substitution process has been performed one more time, that is, next to: $eax_6 := \phi(0, \phi(0, eax_4)+1)+1+5$. Only then would it be clear that recursivity is in play. It could also be said that the $\chi$-function deteriorate the readability, but not to the point of the $\phi$-function.

4.4.2.2 Indirect Memory Accesses

To handle the aliasing issues, the value-set analysis presented in Section 4.3.3 has to be used. As explained, the analysis relies heavily on assumptions about the data layout
of the program being analysed. A program which does not respect these assumptions
will not be able to be correctly analysed [9].

Moreover, the analysis only recovers coarse information about arrays. In the example
presented in Section 4.3.3, the value-set analysis output contained a few $\infty$ for it could
not determine upper bounds. It was only thanks to the analysis performed on edi
that we could find out they were not unbounded. It is then reasonable to think that,
for some programs, or parts of some programs, the lack of knowledge on the variables
would greatly cripple the analysis.
Chapter 5

Conclusion

In the past four chapters, the practice of digital reverse engineering has been discussed at great length. It started with a piece of history, a cornerstone in the computer industry, which showed the relevance of this practice that persisted up to these days. From there, the practice has been gradually described, starting from the technical foundation upon which it lies, up to the latest academical results which are relevant to the contribution of this work.

In this era of digitalisation, mankind finds itself carried away in a constantly increasing reliance on software applications, from mundane activities such as counting footsteps thanks to a pedometer installed on a smartwatch, up to cutting-edge medical analysis provided by artificial intelligence able to understand natural languages. For computer security matters, the tests of correctness of those applications with critical roles, but also because of the perpetual erosion of time which leads source code and documentation to get lost, the need of being able to reason about these applications in their most basic form, that is, as sequences of machine instructions, become apparent. As such, reverse engineering is still a relevant topic as of today, and probably for the many years to come.

As Kevin Coogan and Saumya Debray have said, tools which have been engineered to perform analysis through reverse engineering at assembly level are sparse. They then put forward one new tool to fill this gap, and with it the idea of a potential extension. The contribution of this thesis has then been to explore this idea and to provide a possible way of implementing it.

The original tool was aimed at making assembly code easier to understand by translating it into a functional intermediate representation of type static single assignment form, and by allowing to reason about it thanks to equational reasoning. As it was originally described, it was only able to perform analysis on traces of Intel x86 assembly, making a dynamic analysis tool.
The contribution of this work is to explain how one could allow this tool to also accept assembly listings directly from an executable file, making it usable for static analysis. This seemingly easy task is made complicated by the fact that the original tool has not been released to the public, but also because indirect memory accesses can cause aliasing dependencies which might not be modelised by the intermediate representation. If proper care is not taken when resolving these specific dependencies, the tool will generate erroneous results. As a consequence, most of the complexity in implementing the new tool resides in the pointer analysis and in implementing the original tool which has to be able to reason about the whole Intel x86 instruction set.

In contrary with the tool which only provides dynamic analysis, the proposed one will yield a less readable output. Knowing that one of the main reasons which lead the original tool to be developed was to improve the readability of assembly code, one might wonder if it is worth extending it for static analysis. A decompiler could be the equivalent of the tool for performing static analysis.

Further work on the topic might include implementing the tool, doing researches on more accurate pointer analysis for its precision will increase the readability of the output of the tool, and investigating whether or not optimisation algorithms working on SSA forms would be beneficial for the tool.
Bibliography


[34] Part Guide. Intel® 64 and ia-32 architectures software developer’s manual, 2011.


[43] Civil code section 3426-3426.11.


