An Analysis of x86 Single-Instruction Compiling as an Obfuscation Technique

Lasse Dessau (s142980)
Rasmus Bo Kajberg (s130168)

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

In this thesis we evaluate single instruction compiling as an obfuscation technique. We analyse the inner workings of `M/o/Vfuscator2`, and propose a tool to aid analysis of programs obfuscated by `M/o/Vfuscator2`. We find that `M/o/Vfuscator2` is similar to Virtual Machine (VM) based obfuscation. However, while state of the art VM based obfuscators can construct a custom instruction set for each protected file, this is not the case for `M/o/Vfuscator2`. This is a limitation of `M/o/Vfuscator2`, and not of single instruction compiling in general. Based on this, we conclude that single instruction compiling can be used as an effective alternative architecture in state of the art VM based obfuscators.

Resumé

I denne afhandling evaluerer vi enkelt-instruktionsoversættelse som en obfuskeringsteknik. Vi analyserer de indre funktioner af `M/o/Vfuscator2`, og foreslår et værktøj til at hjælpe med analyse af programmer som er obfuskeret af `M/o/Vfuscator2`. Vi kan se, at `M/o/Vfuscator2` i høj grad minder om Virtuel Maskine (VM) baseret obfuskering. Men mens moderne VM baserede obfuskeringsværktøjer kan konstruere et unikt instruktionssæt for hver beskyttet fil, er dette ikke tilfældet for `M/o/Vfuscator2`. Dette er en begrænsning af `M/o/Vfuscator2`, og ikke af enkelt-instruktionsoversættelse i almindelighed. Baseret på dette konkluderer vi, at enkelt-instruktion oversættelse kan bruges som en effektiv alternativ arkitektur i moderne VM baserede obfuskeringsværktøjer.

Acknowledgements

We would like to thank Christian Damsgaard Jensen, who have been our supervisor during this project, for both inspiring discussions and helpful feedback.

Contents

Introduction

In January 2016, CFCS[1] published a risk analysis of the cyber threat against Denmark in which it stated that the threat of espionage as well as cyber-crime against Danish authorities and companies are very high[Cyb16]. It further stated that both the amount of attempted attacks and the technical sophistication of these have increased in recent years. It is a wide consensus that this is a global trend[Sec15][Ver15].

In recent years malware have been used in cyber attacks against various sectors. Malware being defined as executable code written with malicious intentions[Mica][Bul][Kasa][Sym]. In particular, malware was employed in high-profile thefts against financial institutions, such as the 2015 Carbanak campaign which amounted to an estimated $1 billion in losses[Sec15], and an attack in 2016 against Bangladesh Bank (BB) intending to steal $951 million. Cyber espionage campaigns such as Snake[Kasb] employs malware as a way to get persistent access to targeted networks.

Attackers often need to make malware inconspicuous while infiltrating systems and will therefore obscure it's functionality so that, even if the malware is found, it will delay analysis of the malware and thus development of effective counter measures against it. Techniques to make malware analysis more difficult are generally referred to as obfuscation techniques[Lai16, p. 22].

One such obfuscation technique was described by Christopher Domas at Derbycon 2015[Dom15a], Recon 2015[Dom15b], and DEF CON 23[Dom15c]. In this technique, traditional assembly instructions are reformulated as memory

[1] Center for Cybersikkerhed

and register displacements (`MOV` instructions). Allegedly this makes programs more difficult to reverse engineer as it is difficult to get a grasp of the programs functionality. To exploit this technique, Domas has developed a *mov-only C Compiler*, named `M/o/Vfuscator2`[Doma]. The obfuscation technique was inspired by Steven Dolan[Dol13], who originally showed the `MOV` x86-instruction to be Turing complete in an article published in 2013.

Prior to this project it was unclear how effective this obfuscation technique is, and which analysis techniques and countermeasures can be employed. Domas have identified that decompilation may be a weakness as suggested by his implementation of anti-decompilation post modules[Domb].

Concurrently, parallel with the preparation of this report, related work was published by Kirsch and Jonischkeith that shows that control flow for programs compiled with `M/o/Vfuscator2` can be restored in many cases, even when the obfuscated code is further hardened by the anti-decompilation post modules[KJ16][Jon16].

1.1 The purpose of this report

The objective of this report is to evaluate single instruction compiling as an obfuscation technique by:

1. Researching the purpose of malware analysis.
2. Researching how malware is analysed.
3. Analysing the functionality of `M/o/Vfuscator2`.
4. Determining the strength of `M/o/Vfuscator2`.
5. Researching obfuscation techniques related `M/o/Vfuscator2`.
6. Discussing single instruction compiling as an obfuscation technique based on our findings.

1.2 Method

To examine the purpose of malware analysis we will first describe relevant threat actors and their motivation. This will allows us to understand what the attackers

hope to achieve. Secondly, we will examine how an attack is carried out and how malware analysis can be used in countering such an attack. This should lead to an understanding of why obfuscation is employed by attackers to hinder malware analysis.

We will then describe the process of analysing malware based on Michael Sikorski and Andrew Honig's four phases of malware analysis[SH12]. Having described the four phases we will consider anti analysis techniques for each of these.

Next, we will analyse the functionality of `M/o/Vfuscator2`. We will look at how control flow and binary operations are implemented using only `MOV` instructions. In addition to the obfuscation done by `M/o/Vfuscator2` we will give a brief description of post-process modules that aims to harden the obfuscation in order to prevent simple decompilation.

In order to determine the strength of `M/o/Vfuscator2` we analyse how it affects each of the malware analysis phases. Based on this, we will implement a proof of concept tool that aims to allow analysis of a file obfuscated by `M/o/Vfuscator2`.

Based on our analyse of `M/o/Vfuscator2` we will identify and describe obfuscation techniques that share common traits.

Finally, we will classify `M/o/Vfuscator2` based on the identified techniques in order to discuss the strengths and weaknesses of single instruction compiling as an obfuscation technique.

1.3 About this report

Our work on this report officially began in February 2016. In the course of our analysis of single instruction compiling, we have analysed the inner workings of `M/o/Vfuscator2`. In addition, we have implemented a proof of concept decompiler that is able to reintroduce control flow, call conventions, the stack, function layout, and instruction reduction in movfuscated programs. Finally, we have evaluated the effectiveness of applying the single instruction compiling as an obfuscation technique both relative to other techniques, and how it counters the malware analysis process.

Concurrently, at Recon 2016 June 19, Kirsch and Jonischkeit from Technische Universität München, presented their work on "Demovfuscator"[KJ16], a deobfuscator for `M/o/Vfuscator2`. This program is able to reintroduce control flow, in many cases even if the program is hardened by the post-process modules

Domas implemented. Shortly after the presentation they published Jonischkeit bachelors thesis, that documents some of their work. This thesis was handed in March 2016, and we learned about it when it was published online June 19.

Our work is complementary to work published by Kirsch and Jonischkeit. While they show how post modules for `M/o/Vfuscator2` can be defeated, we focus on furthering the decompilation process.

1.4 Overview of the report

Threat landscape Describes the three categories of threat actors we have identified in association with the use of malware: advanced persistent threat (APT), criminal networks, and hacktivists.

Malware attack anatomy Analysis of how malware analysis can be used in defense against malware based cyber attacks.

Malware Analysis Describes that the process of analysing malware can be split into four phases: basic static analysis, basic dynamic analysis, advanced static analysis, and advanced static dynamic analysis.

Background in x86 Gives a brief refresher of relevant 32bit x86 assembly concepts, the cdecl calling convention, and the ELF file format, which is used in the following chapters.

Movfuscation Details our analysis of how the `M/o/Vfuscator2` works based on public documentation and the source code.

Analysing movfuscated code We look at how `M/o/Vfuscator2` affects the phases of malware analysis, and propose a proof of concept deobfuscator.

Obfuscation techniques Describes obfuscation techniques that have traits similar to `M/o/Vfuscator2`.

Evaluation A discussion and evaluation of the classification and effectiveness of `M/o/Vfuscator2` and single instruction compiling in general.

Conclusion We conclude the evaluation of single instruction compiling as an obfuscation technique and propose future work.

Example Programs and Bibliography A list of source code for the C-example programs and the bibliography cited in the report.

Threat Landscape

The following chapter describes three categories of threat actors that are known to use malware, including their motives and modus operandi.

2.1 State Sponsored Actors

State sponsored actors generally have access to huge amounts of resources as well as cutting edge technologies. A prime example of this is the campaign labeled Stuxnet, that targeted SCADA systems of Iranian nuclear facilities around 2010. Not only did Stuxnet utilize four zero-day vulnerabilities[Mur10], but it was also based on thorough knowledge about the complex systems located in a nuclear facility[KM14, p. 121].

The motives behind state sponsored attacks are often of political nature, aiming at strengthening a nation's political agenda, typically by providing intelligence. This, however, is not always the case as seen with the group labeled APT1 (later PLA Unit 61398), which is said to have targeted large volumes of intellectual property[Man13].

2.2 Criminel Networks

The primary objective of any Cybercrime ring is to earn money. They have the resources to buy the best exploits and malware on the market, or to develop these themselves. An example of such a group is "Carbanak" which has used targeted persistent attacks to steal from around 100 different financial institutions with total looses of close to $1 billion US dollars according to estimations by Kaspersky Lab[Sec15].

Even criminals with limited means (e.g. script kiddies) can buy a malware campaign online as a service[Cyb16]. To market the services at competitive prices, most of the malware obtained by such services, often belong to an already known family of malware.

2.3 Hacktivists

This group is comprised of non-state actors whose primary motivation are political, religious or idealistic rather than monetary. A well known hacktivist groups is *Anonymous* whose goals include combating censorship, promoting freedom of speech, and fighting government control. Anonymous usually attempt to accomplish this through crashing web servers, website defacement and leaking hacked confidential information[Gol][SAN][Pau].

While hacktivists may often be associated with denial of service or defacements, the use malware has also been reported. For example, an attack in 2014 used a fake slide show to deliver malware to ISIS critics with the aim of revealing their IP address along with other information that ISIS could then use to establish their identify [Lai16][SS14].

Malware Attack Anatomy

In this chapter the life cycle of a malware cyber attack is examined, as well as how malware analysis can be an important part of the incident response process used in countering attacks.

There are multiple models used to explain the life cycle of cyber attacks. State of the art models include: The *Cyber Kill Chain* by Lockhead Martin[Eri11][Mar15a], *Attack Lifecycle Model* by Mandiant[Man13], and Dell SecureWorks *APT Lifecycle*[Sec12]. While these models may have been written with Advanced Persistent Threats(APT) in mind, many of the steps are shared with malware attacks by less advanced groups.

The *Cyber Kill Chain* is the most widely cited model and is as expressive as the other models[Sig15, p 13]. In the following sections we describe the *Cyber Kill Chain*, and how malware analysis relates to this model.

3.1 Cyber Kill Chain

According to the *Cyber Kill Chain* a cyber attack consists of seven phases as illustrated in Figure 3.1.

Figure 3.1: Source: Lockheed Martin, Cyber Kill Chain [Mar15b].

Reconnaissance In this phase, attackers research the target[Mar15b] using both passive and active techniques. Passive reconnaissance, e.g. using information publicly available, does not directly interact with the target systems and are therefore especially difficult to disrupt. Active reconnaissance interacts with the victim system and is therefore comparatively easier to detect[J V15].

Information gathered may include which systems are vulnerable to infection, how they are protected, and a selection of possible attack vectors that can be used in the intrusion phase, such as picking potential targets for phishing mails. When the reconnaissance phase is completed, the attacker starts to prepare the actual attack[J V15].

The more information the attacker is able to gather the more likely the attacker is of successfully infecting the target with the prepared malware[J V15].

Weaponization In this phase the attackers prepare for the operation by creating the malware and combining it with an exploit so that it has a greater chance to successfully infect the victim. [Mar15b].

Delivery In this phase the attacker will attempt to get their malware to execute in the target's system. This can be done using a variety of methods

such as email, usb-sticks, social media interactions, or a watering hole attack[Mar15b].

This phase is an important opportunity for the defenders to to block an attack. This can be done by leveraging weaponizer artifacts to detect new malicious payloads at the point of Delivery. Logs should be recording the attack even if the malware remains undetected so that a forensics investigation can later unravel the vector and time of attack[Mar15b].

Exploitation In this phase, the attackers attempts to exploit the system using social engineering or by abusing an unpatched vulnerability[Mar15b]. Typically a successful exploitation will allow the attacker to take control of the victim's infected machine.

The duration of the exploitation phase depends on the chosen attack vector[J V15]. Verizon states that approximately half of the users that are tricked into opening a phishing e-mail and clicking on a malicious link, do so within the first hour[Ver15]. Exploiting a software vulnerability on the other hand, can be almost instant.

Installation In this phase the attacker will attempt to install a backdoor to maintain access, persist the malware by e.g. making it run on boot, and hide the presence by installing rootkits etc.[Mar15b][J V15].

Command and control (C2) In this phase, malware is being connected to the C2 infrastructure of the attacker. For remotely controlled malware this phase is crucial as it allows the attacker to issue commands to the infected hosts.

Actions on objectives In this phase, the attackers attempts to accomplish their mission as they are now ready to start gathering data[Mar15b]. This involves identifying relevant machines and information stores in order to extract data, which can be send out of the victim network, e.g. in small quantities in order to avoid detection[J V15].

This phase may also involve escalating privileges, installing rootkits, tools, backdoors, and compromising laterally [J V15]. Attackers may attempt to continue to hide their presence in order to ensure long term persistence[J V15]. This may involve operating in peak hours to drown their presence in normal legitimate traffic, maintain rootkits, and alter log files to ensure continued concealment.

3.2 Malware Analysis in the Cyber Kill Chain

The goal of the *Cyber Kill Chain* is to understand the phases of a successful attack in order to improve the likelihood of preventing such attacks. The idea is, that if an attack can be stopped at any of the stages before *actions and objectives* phase in the *Cyber Kill Chain*, the attack will not succeed[Mar15b]. The defender can create a matrix describing the course of action required to counter specific phases of the attack[Eri11]. An example of such a matrix can be seen in Figure 3.2, which illustrates which actions that should be taken in regards to each stage and which outcome this action has, such as either detection or disruption of the attack. Many of these actions relies on information gained from malware analysis to be effective.

Phase	Detect	Deny	Disrupt	Degrade	Deceive	Destroy
Reconnaissance	Web analytics	Firewall ACL				
Weaponization	NIDS	NIPS				
Delivery	Vigilant user	Proxy filter	In-line AV	Queuing		
Exploitation	HIDS	Patch	DEP			
Installation	HIDS	"chroot" jail	AV			
C2	NIDS	Firewall ACL	NIPS	Tarpit	DNS redirect	
Actions on Objectives	Audit log			Quality of Service	Honeypot	

Figure 3.2: Source: Lockheed Martin, Course of action matrix [Eri11].

The defenders actions to counter the weaponization, exploitation, installation, and C2 phases relies heavily on the use of intrusion detection(NIDS, HIDS) / prevention systems(NIPS, HIPS)[Eri11].

These technologies in turn relies on indicators of compromise (IOA) and indicators of compromise (IOC), which are derived by analysing behavioral patterns of malware that indicate that a system is infected or under attack, and can be either network based, such as outgoing network traffic to a known malicious domain, or host based, such as the creation of specific files or processes. IOC is typically used for indicators of a system that is already compromised, while IOA

is used for signs of an attacker trying to compromise a system [1][DeC][Eri11].

While malware delivery can sometimes be detected by vigilant users, technologies such as proxies and anti-virus may be able to entirely deny or disrupt the delivery process. Anti-virus requires knowledge about malware or the structure of malware obtained from malware analysis.

If an attack gets by the initial protections, actions should be taken to degrade, disrupt, and eventually destroy the attack. These actions may vary depending on which phase of the *Cyber Kill Chain* the attack is in[Eri11]. The goal is to prevent any further damage, e.g. malware spreading to other hosts or exfiltration of additional data if the attacker has reached the *Actions and Objective* phase[Kra11]. Malware analysis can be used to determine what the malware is attempting to accomplish so that the most effective counter measures can be implemented.

Once knowledge is obtained of how the malware affects the system, the attack can be destroyed e.g. by wiping disks or particular files. During the recovery, the effected systems are brought back into operation, e.g. reinstalling services and removing the containment of the system. It is also important to verify that the system is protected against similar attacks in the future.[Kra11]

3.3 Attribution

Another aspect of malware analysis, not directly linked to the *Cyber Kill Chain*, is attribution.

Attribution is the act of attempting to link the attack to a specific perpetrator. While not impossible, it has proven difficult to follow this up with legal action[Ley15]. However, simply attributing the attack helps in the understanding of the attacker's motives, capabilities and future plans.

One way of connecting two attacks is to look for identical hashes from the malware samples. However, malware samples can be generated so that the hashes will differ, even when the source code of the samples are identical. Other indicators that can give a clue on who is behind the attack could be: compile times that fit with working hours of a specific country or matches compile times of previously seen malware samples, keyboard layouts indicating the language

[1] Occasionally the terms are used as synonyms, simply referring to information that can be used to detect a possible incident

preference of the attacker[Fir13], reoccurring text strings or code reuse[Nov16], and DNS registration information[Fir13].

The target itself might also hint to the attackers origin. An example of this is *APT28*, also known as *Sofacy Group*, who *FireEye*[2] claims are behind a number of attacks sponsored by the Russian government. One of the reasons, that *FireEye* gives for their claim, is that the malware campaigns have been targeting Georgia and other Eastern European governments as well as *NATO*, which they claim fit well with the political interests of Russia[Fir14].

When dealing with attribution however, one should keep in mind, that it is possible to fake some of these artifacts, thereby making it appear as if others are guilty, increasing the difficulty of attributing the attack to the correct threat actor. This kind of false flag operations are a growing trend[Mim16].

3.4 Summary

In this chapter, we have outlined the role of malware analysis in defending against a cyber attack, and have described how it can be used in both a proactive and reactive manner. Specifically, many defensive actions in the *Cyber Kill Chain* rely on information obtained from malware analysis, such as how to detect or destroy a malware attack. Additionally, malware analysis is central to attributing an attack with a specific perpetrator. The process of malware analysis will be explained in more depth in chapter 4.

[2] *FireEye* is a cyber security company, https://www.fireeye.com/

Malware Analysis

In chapter 3, it was argued that malware analysis plays a crucial role in incident handling as it helps determine the extent of damage done by the malware as well as finding indicators of attack (IOA) and indicators of compromise (IOC) to help combat the malware in the future, and potentially attribute the malware to a threat actor.

This chapter describes how malware is commonly analysed. We will later apply this knowledge to analysing movfuscated programs. The chapter is primarily based on the books "Practical Malware Analysis"[SH12] and "Malware Analyst's Cookbook"[MR11].

4.1 Malware Behaviour

When analysing malware it is essential to know what you are looking for. In this section we will categorise the most common types of malware and their behaviours[SH12, p. 3]. In practice, malware can have traits from several of these categories[SH12, p. 4], but knowing the goals and tactics of typical malware categories makes it easier to recognise malicious behaviour.

Worm / Virus While a worm and a virus might behave differently, what they have in common is that they attempt to spread the infection[SH12, p. 4].

Backdoor A backdoor gives the attacker access to the infected system, typically by providing a shell[SH12, p. 3].

Botnet A botnet is similar to a backdoor, but whereas the backdoor allows the attacker to issue commands to individual targets, the same command will usually be send to all of the targets that are a part of the botnet[SH12, p. 3].

Downloader / Droppers This is a category of malware that are used for downloading (dropping) additional malware on a target system[SH12, p. 3].

Launcher This type of malware is used to launch other malware stealthly or with greater privileges[SH12, p. 4].

Ransomware A ransomware prevents victims from accessing either their system or files until a ransom is paid. This is usually done either by denying system access or by encrypting user data. In order to regain access, the victim has to pay the attacker a ransom[Micb].

Information stealing malware A wide variety of information can be of interest for the attacker. Some malware is designed to steal information such as keystokes, images and the like[SH12, p. 4].

Spam-sending malware Sending spam is a way to earn money, but it requires a certain number of resources. Because of this some malware are created with the purpose of distributing spam messages[SH12, p. 4].

Scareware This category of malware tries to scare the victim into paying money[SH12, p. 4], e.g. by claiming that the victim is infected with a virus that can be removed with some fake anti-virus software, or by claiming that the victim have broken a law and has to pay a fee[Inv].

Rootkit The purpose of a rootkit is to hide activity and remain persistent without being detected[SH12, p. 4].

4.2 Defining Obfuscation

In general, *obfuscation* is defined in Oxford Dictionary as: “to make obscure, unclear, or unintelligible”[Dic].

Within the field of computer science, Barak et al. further defines *code obfuscation* as: “Informally, an obfuscator O is a compiler that takes a program P as input and produces a new program O(P) that has the same functionality as P yet is "unintelligible" ”[Bar+01].

That is, obfuscated programs must make code unintelligible while retaining computational equivalence with the original program, that is exhibiting the same computational effect[BB14, p. 267-268].

Barak et al., however, shows that an ideal obfuscation resulting in complete unintelligibility is impossible[Bar+01], so practical code obfuscation must content with making functionality more difficult to analyse - hindering automatic analysis and making manual analysis more time consuming[SH12, p. 327].

The motives for obfuscation extend beyond keeping the functionality of malware hidden, being applied legitimately in digital rights management to protect intellectual property[BB14, p. 267].

4.3 Static and Dynamic Analysis

Literature often defines two approaches to malware analysis: *static* and *dynamic* analysis[SH12, p. 2][MR11, p. 283].

Static analysis attempts to uncover information about a program without executing it while dynamic analysis monitors and logs a programs behavior during execution.

Generally speaking dynamic analysis is more time efficient and can to an extent be automatized[Zel16] but does not guarantee that the the full functionality of the program is uncovered. There is also an inherent risk involved with dynamic analysis, since it requires executing the malware, although this can be mitigated by isolating it in a protected environment such as a sandbox[SH12, p. 29].

Static analysis on the other hand can uncover the full functionality of the malware[SH12, p. 3] but can require significantly more time and/or disassemble skills[MR11, p. 283][Zel16].

Sikorski and Honig further divide static and dynamic analysis into *basic* and *advanced* variants[SH12, p. 2-3].

One approach to analysing malware could be the four step process shown below in Figure 4.1 which combines basic and dynamic analysis to efficiently uncover information early in the process. If enough information is uncovered during the basic phases then the advanced analysis may not be required.

Figure 4.1: Malware analysis pipeline

In practice, a malware analyst may reiterate or reorder the phases depending on the analysis. The following subsections gives a brief explanation of each of these phases.

4.3.1 Basic Static Analysis

The purpose of basic static analysis is to uncover properties and information about a program without executing it. As illustrated in Figure 4.2 the first step in doing so is to fingerprint the file to determine whether it is a known file. This could include scanning the file using an antivirus product. If the file is recognised as previously seen malware, the malware may already have been analysed sufficiently by other parties. If the file is not recognised or if more details are needed, the analyst can gather and analyse further artifacts such as:

File signature A fingerprint of a malware can be used to search for matching signatures, i.e. in VirusTotal[1] or other databases, to see if someone else have already done analysis on an identical sample. Antivirus solutions also make use of signatures to identify known malicious software[SH12, p. 10].

Strings Any strings that appear in clear text can easily be extracted from the file. Strings such as suspicious URLs can give a hint of the programs functionality[SH12, p. 11-13].

Packing Programs can be packed for legitimate reasons such as compression and copy protection, but packing is also frequently used by malware authors to hinder malware analysis. Entropy analysis[SH12, p. 387] and packer detection tools[SH12, p. 13-14] can statically check for packed data.

File headers and sections Can contain information such as: whether the program is console or window based, which compiler build the program, and when the program was build. File headers and sections can also provide hints to whether the program is packed or not[SH12, p. 22-24].

[1] https://www.virustotal.com/

Loaded libraries Imports tells us which functionality the program can use from third party libraries [SH12, p. 15-17]. This may allow a malware analyst to make hypotheses such as whether the malware logs keyboard presses, uses encryption, and/or connects to the Internet.

At the end of this phase the analyst should have a basic idea of the functionality of the file, however this depends on whether the file is packed or not, and further analysis is often needed[SH12, p. 26].

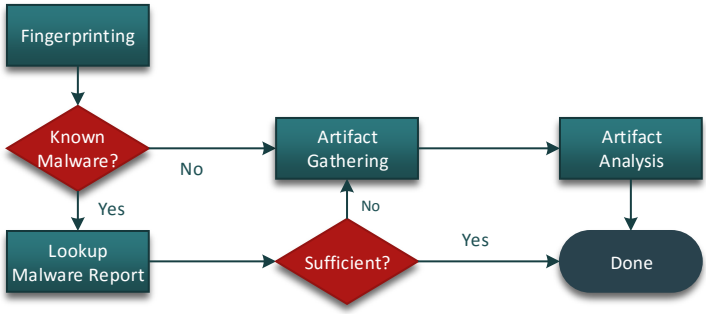


Figure 4.2: Basic static analysis

4.3.2 Basic Dynamic Analysis

An efficient method for determining a programs functionality is to simply execute it while monitoring its behavior and how it affects the environment. Typical the following is worth monitoring:

File activity It may be interesting to observe which files are read and written to by a malicious program. This may hint to the purpose of the malware, e.g., a ransomware encrypts a huge number of files and a dropper downloads additional software.

Registry activity On Microsoft Windows the registry is a common way for malware to gain persistence[SH12, p. 241], e.g., by adding the malware as a value to one of the registry keys that controls startup applications such as `"HKLM\Software\Microsoft\Windows\CurrentVersion\Run"` [Car11, p. 40].

Network activity Network packet data may give a clue to the purpose of the program. Network traffic can include IP-addresses and ports that can be used as an IOC.

Process activity API functions such as ones that load additional libraries, install hooks, or run commands on the system may also give clues about the purpose of the program.

Figure 4.3 shows the process of basic dynamic analysis. Initially a safe lab environment with monitoring capabilities is prepared. Then the malware is run while the lab records it's behaviour. In the next step the logs generated from the monitors are examined.

If information indicates that the sample may run differently under alternate conditions, then it may be necessary to rerun the test in an adjusted lab environment. This could, e.g., include setting up a fake host if the initial analysis revealed network connection attempts by the executed file.

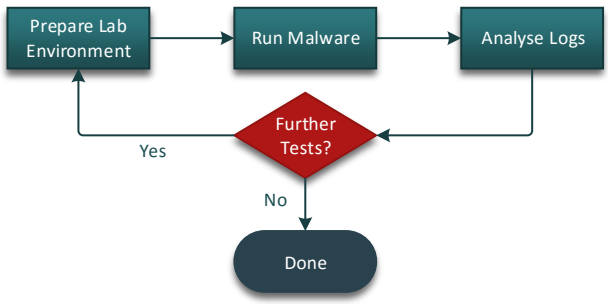


Figure 4.3: Basic dynamic analysis

The advantage of this approach is that even if the malware is obfuscated making a static approach difficult, executing the malware will still exhibit it's malicious activity.

A difficulty with dynamic analysis is that the execution environment may not trigger the full functionality of the malware, e.g., if the malware is aware that it is running in a sandbox[SH12, p. 36-37] or is programmed to only attack under specific circumstances such as ransomware avoiding schools and hospitals[GL16].

4.3.3 Advanced Static Analysis

Advanced static analysis is a set of techniques to uncover a programs functionality by examining the code of a compiled program. The first step, as shown in Figure 4.4, is usually to disassemble or decompile the program into a more readable format. Programs such as IDA Pro[2], Hopper[3], OllyDbg[4], and Radare2[5] will help visualize the dissambly and flow of a compiled program, and some can even decompile code into pseudo C language syntax.

At this point a malware analyst can tell whether the program is packed or not by examining the decompiled code. If the code is packed then it may be unpacked by using an existing tool, or by statically reversing the unpacking algorithm, or by applying advanced dynamic analysis techniques which will be described in section 4.3.4. If the code is further obfuscated then the analyst may focus on de-obfuscation before continuing the actual analysis. During the final analysis phase the analyst typically attempts understand how the program works and what it's purpose is.

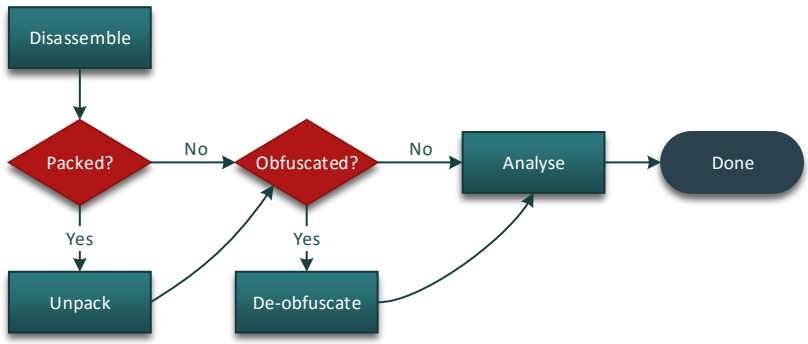


Figure 4.4: Advanced static analysis

Advanced static analysis is difficult to automate. It requires both significant effort and is considered more difficult than basic static and dynamic analysis[Zel16]. However, in contrast to the basic analyses, it is possible to reveal the full functionality and identify all possible execution paths that the program

[2]https://www.hex-rays.com/products/ida/
[3]http://www.hopperapp.com/
[4]http://www.ollydbg.de/
[5]http://radare.org/r/

may take. This may for example reveal hidden functionality that is triggered only in certain execution environments or using specific arguments.

4.3.4 Advanced Dynamic Analysis

Advanced dynamic analysis is similar to basic dynamic analysis, illustrated by Figure 4.3. However, instead of just examining the resulting logs after execution, the process is debugged, e.g. in order to observe transitional states and/or force the program to take alternate execution paths.

An interesting transitional execution state in a packed malware is the point where the unpacking algorithm is done, leaving the unpacked program code in memory. At this point the unpacked malware can simply be dumped for further static analysis.

4.4 Anti Analysis

In chapter 3, it was explained how malware analysis is useful in defence against attackers and in attributing an attack to a perpetrator. This gives malware authors an incentive to make analysis more difficult by introducing obfuscation techniques.

Different techniques exists and have a varied effect on each phase of the analysis, e.g. some techniques effects basic static analysis while having no effect on the other phases. In some cases however, a technique can have an impact on several phases, or be combined with other obfuscation techniques to increase the resilience against analysis.

For the purpose of this report we divide obfuscation techniques into four "anti"-groups each targeting one of the four different phases of analysis.

4.4.1 Anti Basic Static Analysis

Obfuscation techniques that target basic static analysis deals with hiding artifacts that can be extracted statically from the malware, such as text strings and imported/exported functions. There are several ways to do so, including simple encoding of strings or, in more advanced cases, the use of packers.

Additionally, attackers can ensure that the signature of their malware is unique so that it is not immediately recognized by signature based solutions, such as anti-virus or virus databases. Signatures can be modified by either manually modifying the malware-file or using a polymorphic encoder such as the Shikata Ga Nai encoder provided by the Metasploit Framework, which create as many unique versions of the same malware as needed[Rap].

4.4.2 Anti Basic Dynamic Analysis

Since basic dynamic analysis relies on running the malware in a safe and monitored environment, a simple way to thwart basic dynamic analysis is preventing the malware from running at all. Malware, however, still need to run to serve it's purpose. So the the malware author wants to make an educated guess on whether or not the malware is being executed by the targeted victim or by a malware analyst. Since malware analysts often use virtual machines as labs, the presence of artifacts associated with virtual machines could indicate that the malware is being analysed. If the malware detects such artifacts it can alter its execution, e.g., to appear harmless, terminate, or delete itself.

Detecting artifacts related to virtualization used to be a common way to do this, however, this is a fading trend since virtualization is becoming more popular in production environments[SH12, p. 369-370].

When executing a potential malware and recording its actions, the analyst has to make a decision on how long to run the malware based on whether or not enough of the programs functionality has been revealed. This can be exploited by malware authors by making the malware sleep for some time in the hope that the analysis is done by then[Ins]. Once the timer runs out, the malware will start its malicious activity.

4.4.3 Anti Advanced Static Analysis

Anti advanced static analysis techniques aim to make analysing the disassembly of malware more difficult and time consuming. While there are many techniques that accomplish this, many of them attempts to either defeat conventional program analysis algorithms, increase complexity of the disassembly, or by tricking disassembler to produce a misleading instruction listing[SH12, p. 227]. A number of anti advanced static analysis techniques will be described in chapter 8.

4.4.4 Anti Advanced Dynamic Analysis

Techniques that target advanced dynamic analysis aim to disrupt the use of a debugger. Malware can attempt to detect if it is being debugged, and alter its behavior depending on the result, e.g. quitting or simply behaving differently in order to thwart the analysis.

While debugging a program an analyst will often single step through the instruction sequence of the malware at a rate slower than normal execution speed in order to ascertain the programs behavior. Malware commonly exploit this fact by timing execution speed, and alter it's behavior if the code runs slower than expected[SH12, p. 357].

Background in x86

In the following sections we briefly describe background theory that will later be used in our analysis of `M/o/Vfuscator2`.

5.1 The x86 Architecture

The instruction set family x86 is based on Complex Instruction Set Computing (CISC) design. In this report we focus on the 32 bit version of the architecture. The 32 bit architecture have eight general purpose registers, two of which are dedicated to hold the *stack pointer*(`ESP`) and the *base pointer*(`EBP`)[Int16].

5.2 Assembly in x86

Instructions can have zero, one, or two operands. In the Intel syntax, which is used throughout this report, the destination operand is specified before the source operand. The x86 instruction set allows for complex 32 bit addressing modes such as the following instruction, that copies 4 bytes (a double word) from the address ($\texttt{EBX} + \texttt{ECX} * 4 + 0x804ff0$) into the `EAX` register[Int16].

```
1 mov eax, DWORD PTR [ebx + ecx*4 + 0x804ff0]
```

5.3 Stack Frame and Call Convention

The EBP register conventionally holds a base pointer to the start of the stack frame. Assuming a cdecl call convention, memory just above EBP holds the function return address and any arguments to the function. Memory just below holds local variables. Memory at EBP holds the EBP of the callee.

In the cdcl call convention arguments to a function are pushed onto the stack before it is called[Fog04]. The following piece of code shows how "Hello, world!" might be written in x86 assembly:

```
1 push 0x804fff1 ; "Hello, world!"
2 call 0x804be01 ; <puts@plt>
```

A called function pushes the callee's EBP to the stack, and then sets EBP to ESP. ESP is then decreased so that the stack frame can contain any local variables used by the function. The result of this prologue is a stack frame similar to the one illustrated in Figure 5.1.

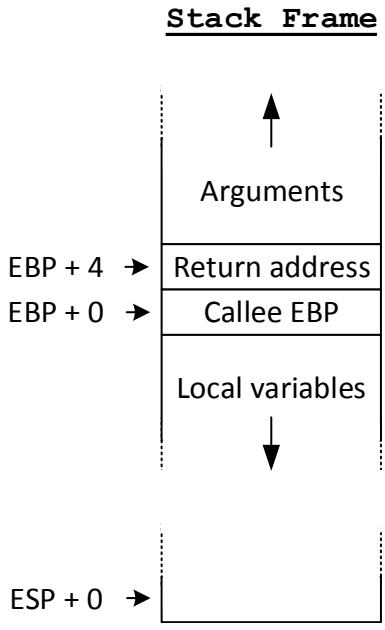


Figure 5.1: Illustration of a stack frame after the prologue.

The callee's EBP and ESP is restored before returning, e.g., by using LEAVE.

```
1 push ebp
2 mov ebp, esp
3 ...
4 (function body)
5 ...
6 leave
7 ret
```

5.4 The ELF File Format

An ELF file contains a number of segments. Commonly .text stores code, .data stores initialized data, and .bss contains uninitialized data[Com95]. Other segments exist, but will not be discussed any further in this report.

Movfuscation

The first part of this chapter introduces the research that lays the foundation for single instruction compiling. The second part focuses on our analysis of the implementation of the single instruction compiler `M/o/Vfuscator2` developed by Domas[Doma]. The description of the inner workings of the `M/o/Vfuscator2` is the result of our analysis based primarily on the open-source code published by Domas[Doma] and secondly on the information given by Domas in talks at various conferences[Dom15a][Dom15b][Dom15c].

6.1 Single Instruction Compiling

The term, *one instruction set computing* (OISC), is defined as an architecture that exhibit Turing completeness using only a single instruction[NWH04]. The Ultimate Reduced Instruction Set (URISC) was suggested by Mavaddat and Parhami[MP88]. Their main idea was to use only a single instruction with the following semantic: "Subtract and branch on less than or equal to zero"[NWH04]. The instruction's combination of subtraction and jump semantics was shown to be sufficient for Turing completeness[MP88].

Jones showed in 1988 that a Turing complete OISC could be formulated based on a move instruction. This idea, however, required extra hardware in the form of memory mapped computational units[Jon88]. That is, functionality is gained by writing memory to triggering ports, i.e., specific memory locations. In effect this is a Transport Triggered Architecture (TTA)[Dol13].

Real world machines designed on OISC principles have been build, such as the MAXQ TTA family of processors by Maxim Integrated[Dol13][Int08].

In this report we focus on the x86 family of Complex Instruction Set Computing (CISC) architecture. Although the architecture specifies many different instructions, the ideas behind OISC can be applied in order to compile programs only consisting of a single type of instruction. In particular, Dolan showed in 2013 that the `MOV` instruction is sufficient for Turing Completeness, even when disallowing self-modifying code and without transport-triggered calculations[Dol13]. Since x86 `MOV` is Turing complete, every other x86 instruction that is able to emulate the behavior of `MOV` must in itself be Turing complete. Domas have used this corollary to show that the following x86 instructions also are Turing complete: `XOR`, `SUB`, `SBB`, `ADD`, `XADD`, and `ADC`[Domb].

While the original motives for developing OISC ranges from educational purposes[Jon88][MP88] to increasing cache size by reducing the silicon currently dedicated to complex functional units[Dol13], Domas suggested the use of Single Instruction Compiling as a way to obfuscate program code[Dom15a].

6.2 The Movfuscator

`M/o/Vfuscator2` is a 32 bit C to `MOV`-instructions compiler in active development by Christopher Domas[Doma]. The compiler is a fork of Little C Compiler (lcc) written by Chris Fraser and David Hanson [FH03].

Figure 6.1 shows how a programs main function is obfuscated by `M/o/Vfuscator2` from just a few C lines into 510 lines of assembly. The `MOV`-instruction sequence, in contrast to regular x86 assembly, have no obvious branches or calls. Also in contrast, GNU Compiler Collection (gcc), a regular optimizing C-compiler, generated just 16 instructions for the main function.

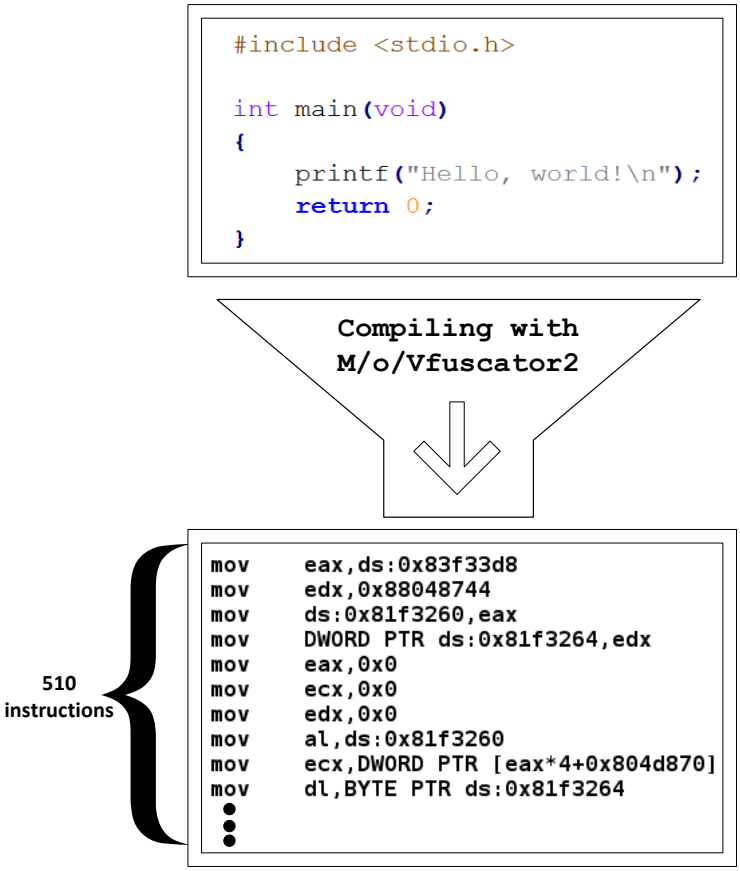


Figure 6.1: Compiling with the movfuscator increases the size of the main function from a few lines to 510 instructions.

6.3 A Backend for LCC

Compiling a program typically consists of several phases, such as lexical and syntax analysis, type checking, and register allocation[Mog11] to name just a few.

A compiler typically have at least a separate front- and backend. The frontend is responsible for checking that the given input conforms to the input language

specification and for producing an intermediate representation that can be used by one of the compiler backends. Each target architecture usually has it's own backend which is responsible for transforming the intermediate representation to actual emited assembly code. Assembly code is usually compiled and linked into a working executable by separate assembler and linker programs.

`M/o/Vfuscator2` reuses the phases implemented by lcc, but the code generation phase have been altered so that the emitter outputs only `MOV`-instructions.

6.4 Compiling C to MOV

A major challenge solved by Dolan and Domas was how to transform higher level constructs into sequences of `MOV` instructions[Dom15b][Dol13]. The compilation of a high level language like C to `MOV` instructions requires support for at least the primitives listed in table 6.1:

Types of primitives that must be translated include	
Control flow	**Binary operations**
Internal function calls	Arithmetic operations
External library calls	Bitwise operations
Conditional branching	Bit Shift operations
Unconditional jumps	Boolean operations

Table 6.1: Primitives that must be compiled to `MOV` instructions

With translations for these primitives in place more complex control flow constructs such as `if/else`, `while`, `for`, and `do` can also be compiled to `MOV` instructions. In x86 assembly MOV cannot be used to alter the instruction pointer (`EIP`) directly (i.e. it's impossible to emulate jumps by moving the target address to `EIP`). This means that a list of `MOV` instructions will always execute in linear fashion and then halt. Our analysis of how Domas, based on Dolan's research[Dol13], have overcome this problem and thus implemented Control Flow will be presented in section 6.5. Suffice to say that the solution requires two separate virtual stacks allocated in static memory which adds several megabytes in the default configuration to the programs memory size as shown in table 6.2.

Another problem with Compiling C to `MOV` is that `MOV` cannot be directly used to compute the result of binary operations[Jon88]. While Jones solve this problem by introducing Transport Triggered Actions (TTA), Domas instead use lookup tables with precalculated values to accomplish the same feat[Domc, lines 509-898]. This will be explained further in section 6.6. The lookup tables used for binary operations further increases the program memory size as shown in table 6.2.

The total static memory size of a small movfuscated program such as the example program in listing A.1 is with the default movfuscation configuration at least 8 MB. Since ∼2 MB stack is placed in the `.bss` section which contains uninitialized data the physical file size of a movfuscated program is ∼6 MB, since the `.bss` section occupies no file space until the program is run[Com95, p. 1-15]. If the program is not stripped of debugging symbols the physical file size increases by ∼4 MB to at least ∼10 MB. In contrast C-programs compiled by gcc can be as small as or even less than ∼10 kB.

Note, however, that the default `M/o/Vfuscator2` configuration aims to strike a balance between size, features and execution speed. For example binary float operations is not part of the default `M/o/Vfuscator2` configuration and enabling it requires significant additional space for lookup tables[Doma]. On the other hand 16 bit arithmetic tables is part of default `M/o/Vfuscator2` configuration even though 16-bit tables can emulated using 8-bit tables although at the cost of an increase in total number of `MOV` instructions and thus execution speed[Domc, line 605, 610, 1233].

Segment	Type	Size
.data	Virtual stack	∼2 MB
	16bit arithmetic tables	∼2 MB
	Bitwise operation tables	∼1 MB
	Multiplication and division tables	∼1 MB
	Arithmetic tables	∼267 kB
	Utility tables	∼67 kB
	Single bit manipulation tables	∼10 kB
	Virtual states	<1 kB
	Boolean truth tables	64 bytes
	Virtual registers	40 bytes
.bss	Virtual discard stack	∼ 2 MB
Total size:		∼8 MB

Table 6.2: Memory layout of a movfuscated program

6.5 Control Flow

Subsection 6.5.1 explains how movfuscation solves the problem of jumping by implementing a way to selectively execute instructions as though they had no effect. The subsections that follows focuses on how this is used to implement high level control flow constructs.

6.5.1 Toggling execution

One of the requirements for Turing completeness is that the architecture must allow for non-termination[Dol13] i.e. be able to emulate conditional unbounded loops. This in turn requires the ability to execute instructions multiple times. In his paper, Dolan originally used a single jump instruction at the end of the instruction list back to the programs first `MOV` instruction. Domas instead found a way to rerun the instruction sequence by triggering an exception using an illegal `MOV` instruction as illustrated in figure 6.2. The instruction used for this purpose is `mov cs, ax` [Domc, line 3090] which is illegal in x86 and therefore triggers an exception. In the prologue of the program, an exception handler have been created to handle this specific exception by restarting the program. The program's instruction listing can thus be seen as one big *master loop*.

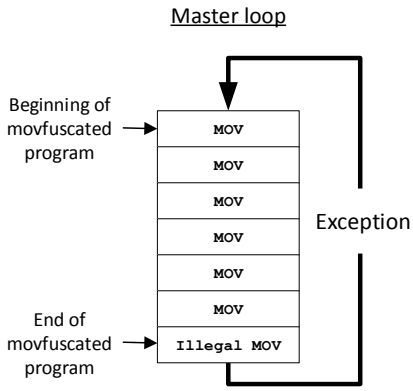


Figure 6.2: A movfuscated program is a sequence of linear executed `MOV` instructions that are repeated until the program halts.

Since the only way to get from address X to address Y is to run all instructions in-

between, the emulation of branching requires a way to execute these in-between instructions without them having a computational effect as illustrated by figure 6.3.

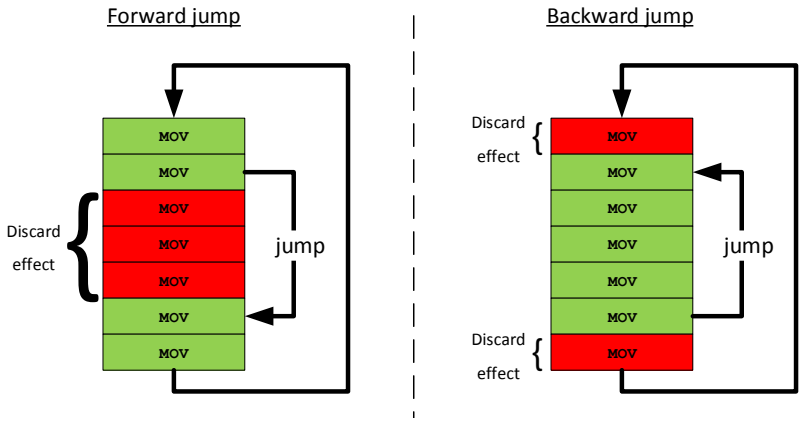


Figure 6.3: Since there is no way to skip `MOV` instruction, jumps are implemented by discarding the effects of in-between instructions.

To discard the effects of executing in-between instructions, Domas introduced the notions of toggling the discard state as `execute on` and `execute off`. When the program is in the `execute on` state, instructions write to a normal *virtual stack*. In contrast when the program is in the `execute off` state, instructions write to a decoy *virtual discard stack*. The *virtual registers* are preserved by pushing them to the normal *virtual stack* whenever execution is turned off, so that when execution is turned on again the register can be restored simply by popping them off the *virtual stack*.

Specifically, the implementation store the execution state as a global variable with value either `ON=1` or `OFF=0`. This value is then loaded into `EDX`. Since addresses are double words of 4 bytes the following assembly line will select a pointer to either the *virtual stack* or the *virtual discard stack* from the two-entry table *sel_data* depending on the value of the execution state:

```
1 mov eax, dword ptr [sel_data + 4*edx]
```

Consider figure 6.4. The instruction `MOV` following `Select data` may attempt to

modify the stack. This is allowed since the execution state is on. In particular, the `Select data` will move the execution state (`ON=1`) to `edx` and then resolve the above assembly to `mov eax, dword ptr [sel_data + 4*1]`. This moves a pointer to the normal *virtual stack* into `EAX`, so that `MOV` instruction that follows `Select data` will access the normal *virtual stack* rather than the *virtual discard stack*.

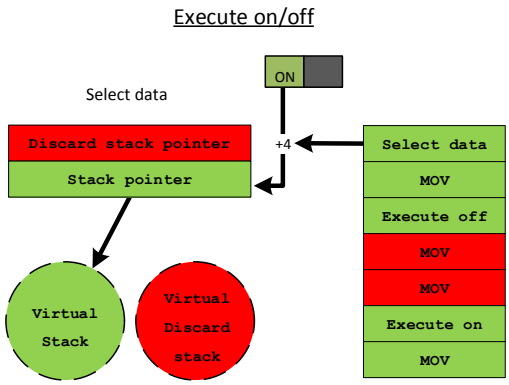


Figure 6.4: A movfuscated program in the `execute on` state is allowed access to the virtual stack.

In figure 6.5 the execution state is off (`OFF=0`). In particular, the `Select data` will now instead resolve `mov eax, dword ptr [sel_data + 4*0]`. This moves a pointer to the *virtual discard stack* into `eax`, so that the normal *virtual stack* will remain unaffected by `MOV` instructions that follows.

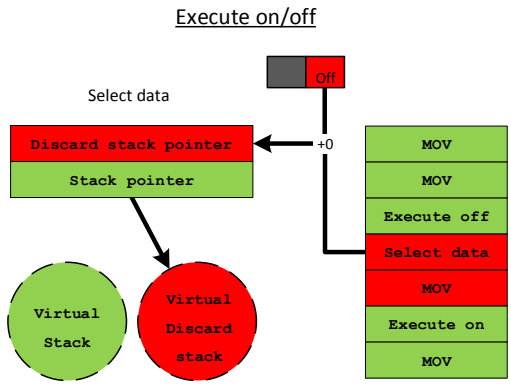


Figure 6.5: A movfuscated program in the `execute off` state is denied access to the normal *virtual stack* and instead use the *virtual discard stack*.

6.5.2 Unconditional jumps

The unconditional toggling of execution state so far presented does not allow for nested jumps since it must be possible to jump over in-between instructions even if they contain `execute on` primitives. Domas handles this by introducing a target jump address. In particular, `execute off` now stores the destination address. Each time an `execute on` is reached the target is compared with the address of `execute on` (EIP). If target does not match EIP then the `execute on` is discarded by forcing it to write to the *virtual discard stack* rather than the *execution state* global variable as illustrated in figure 6.6.

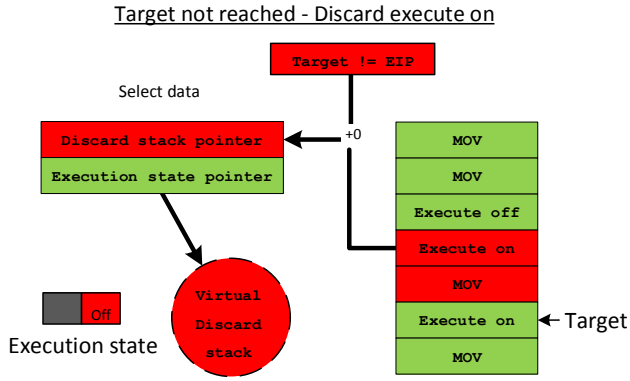


Figure 6.6: A movfuscated program in has yet to reach the jump target so `execute on` is discarded by writing 1 to the *virtual discard stack* instead.

In figure 6.7 the target has been reached so *execute on* is allowed to write through to the *execution state* global variable thus enabling execution and thus completing the unconditional jump.

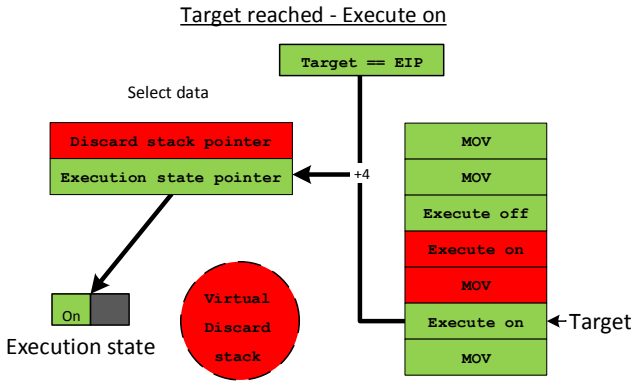


Figure 6.7: A movfuscated program has now reached the jump target so `execute on` is allowed to write through to the *execution state*.

6.5.3 Conditional branching

Conditional branching is implemented as unconditional jumps, as described in the previous section, except execution is only turned off if the branching condition is met. The boolean condition is evaluated to either 0 or 1 and stored in `B0`. Next B0 is used to select either the pointer to the *execution state* or *discard stack*.

Figure 6.8 illustrates a conditional jump. Since the branching condition is met, B0 is 1 and the *execution state* is set to off, thus emulating the jump.

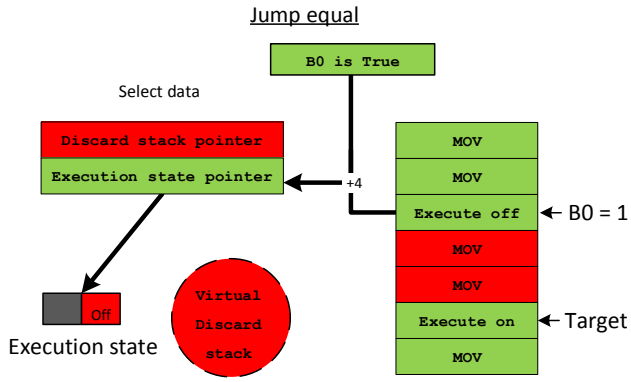


Figure 6.8: A movfuscated program toggling the execution state to off.

6.5.4 Internal call

An internal call works similar to an unconditional jump, but before execution is turned off the return address is pushed to the *virtual stack*. This address is later used to make another unconditional jump from the end of the called function back to the callee.

`M/o/Vfuscator2` uses the call convention cdecl[Domc, line 82]. Arguments are pushed to the *virtual stack* and the caller is responsible for cleaning up the arguments off the *virtual stack* when the call terminates.

6.5.5 External call

While internal calls relies on conditional `execute off` primitives, linked external library functions can only be invoked using an actual jump.

To get around this limitation Domas introduced another exception handler to handle external calls as shown in listing 6.1. This exception handler can be invoked by a `MOV` instruction that access an illegal memory address and thus generates a `SIGSEGV` signal. Before the fault is generated the return address is pushed to the *virtual stack* so that the `RET` instruction of the external function will return control to the callee at the correct address.

External calls expects the arguments and the return address to be located on the *real stack* as opposed to internal functions that use the *virtual stack*. Listing 6.1 shows how this is facilitated by moving the virtual stack pointer into *ESP*.

```
1 mov esp, DWORD PTR [virtual_stack_pointer]
2 jmp DWORD PTR external_function
```

Listing 6.1: Dispatch handler for external calls

6.6 Binary Operations

While `MOV` is a Turing complete instruction, it was not designed to perform non-memory arithmetic's. To tackle this issue Christopher Domas developed an arithmetic logic unit (ALU) for `M/o/Vfuscator2`.

Since the different binary movfuscasted operations are based on roughly the same idea, we present only our analysis of addition. The main difference is the dimension and size of the lookup tables.

6.6.1 The ALU

The ALU makes it possible to translate regular arithmetic assembly instructions - such as addition (`ADD`) - into series of `mov` instructions that accomplish the same computation using memory access arithmetic.

To implement different arithmetic operations the memory contains a number of lookup tables that can be referenced by the `MOV` instructions.

The ALU has four global variables for communicating input and output to the movfuscated program. These are `alu_x` (operand 1), `alu_y` (operand 2), `alu_s` (result), and `alu_c` (carry/overflow).

6.6.2 Addition

To perform addition, the Movfuscator takes advantage of address arithmetic supported by the `MOV` displacement encoding. Displacement is used to move data from or to a specific index relative to a base pointer (such as when accessing an array). The C command $dest = base[x]$ for a 32bit integer array is written as the following in assembly:

```
1 mov dest, dword ptr [base + 4*x]
```

The `MOV` instruction requires computed addresses to be a valid accessible memory segment. Furthermore, for arithmetic to work the memory must contain tables with specifically crafted values. `M/o/Vfuscator2` compiles into static memory the two-dimensional lookup table for addition shown in table 6.3.

X/Y	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	5
2	2	3	4	5	6
3	3	4	5	6	7
4	4	5	6	7	8

Table 6.3: Lookup table *alu_ add* for addition.

With this two dimensional table in place, the following `MOV` instructions will add the numbers in register x and y and store the result in register *res*, that is we compute $res = x + y$.

```
1 mov esi, dword ptr [alu_add + 4*x]
2 mov res, dword ptr [esi + 4*y]
```

The values of x and y registers are used as coordinates in the lookup table. In particular, the first instruction computes a pointer to the *X-column*. The second instruction then displaces this pointer along the *Y-row* by the value of

y dwords in effect adding x to y. The instruction sequence also corresponds to the following command in C-syntax: $res = alu\_add[x][y];$.

The same guiding principle is used for the other binary operations, but with lookup tables containing results depending on the type of operation e.g. subtraction.

6.6.3 Space efficient implementation

In practice the two full lookup tables needed to do a binary operation such as 32 bit addition would require $2 \times 4 \times 2^{32} bytes \simeq 34 GB$ of memory. The solution implemented by `M/o/Vfuscator2` is to instead connect two 16 bit adders as shown in figure 6.9. In contrast, with a 1-bit carry, 16+1 bit lookup tables requires only $2 \times 4 \times 2^{16+1} bytes \simeq 1 MB$.

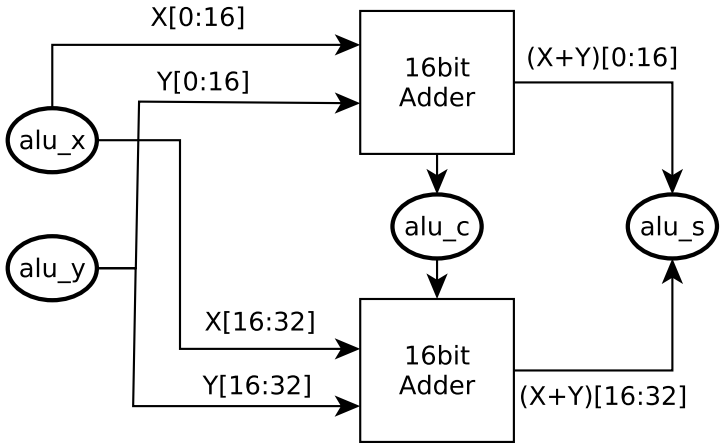


Figure 6.9: 32-bit addition using two 16-bit adders

The use of two 16 bit adders increases the complexity of the the assembly code but the fundamental idea remains the same. The final result is stored in `alu_s`. Notice how the first 16-bit adder stores the result in lower 16-bit of `alu_s` and the second adder in the high 16-bits of `alu_s`. A copy of the 17bit result from the low 16 bit adder is copied into the 32bit carry field `alu_c`. The high 16 bit adder retrieves the high 16 bits of `alu_c` and adds it to the sum.

6.7 Post-process Scripts

In addition to the main component of the `M/o/Vfuscator2`, Domas have created a number of post-processor scripts that works on code first compiled with the `M/o/Vfuscator2`[Domb]. The scripts can be divided into two categories: scripts that translate `MOV` instructions to other x86 Turing complete instructions, and scripts that hardens the `MOV` output of the compiler by layering additional obfuscation techniques.

Peephole Shuffler This technique identifies neighboring `MOV` instructions that does not influence eachother and thus can be swapped without changing the computational effect of the program[Domd]. The algorithm by default iterates over the instruction 10 times.

The effect of this post module is the permutation and interleaving of `MOV` aimed at countering simple decompilation through pattern based matching [Doma].

Register reallocator This technique shuffles registers of `MOV` instructions to make pattern based decompilation harder. However, since the basic flow and layout of instructions remain the same this may be less effective than the *Peephole Shuffler* on its own.

While these techniques may be effective against pattern based decompilation, a recent study, published online during our work of analysing the `M/o/Vfuscator2`, shows that this can be overcome by conducting taint analysis[Jon16]. In addition, Domas have stated himself that the post modules should be seen as merely a proof of concept or a starting point from which to develop more effective countermeasures against decompilation[Domd].

Analysing Movfuscated Code

In this chapter we analyse a movfuscated program using the four phases of malware analysis presented in chapter 4. Finally, we present our proof of concept deobfuscater.

7.1 Example Programs

For this chapter we have written two programs shown in listing A.2 and A.3. The first program acts as a backdoor by attempting to establish a reverse shell to a list of IP's. The first program also forks to give the illusion that it terminates immediately. The later program is an extension used the dynamic analysis. In particular, the later program logs the commands used to establish the reverse shells to a file.

7.2 Basic Static Analysis

In this section we do a blackbox basic analysis of the example program in listing A.2 compiled by the `M/o/Vfuscator2`.

File signature Multiple compilations of the same program results in identical

hash values unless one of the randomized post modules is also explicitly applied. If a malware campaign relies solely on `M/o/Vfuscator2` without applying the post-process modules, signature based scanners will be able to identify and contain the malware as soon as a single sample have been analysed.

Strings Neither the `M/o/Vfuscator2` or any of the post modules obfuscate constant strings. Strings are therefore clearly visible as shown in figure 7.1.

```
r@debian:~/Desktop/mal$ strings suspicious | head -n 20
/lib/ld-linux.so.2
libc.so.6
libm.so.6
sigaction
exit
sprintf
system
sleep
fork
GLIBC_2.0
192.168.100.3
192.168.100.2
192.168.100.1
nc -w 1 -e /bin/sh %s %d
```

Figure 7.1: Looking at the strings in a movfuscated files can still reveal info about its functionality.

Packing The entropy of the `.text` section which contains the instruction sequences of the obfuscated example program is computed using "`binwalk -entropy`" to ~ 0.6. The same program compiled using gcc produces a `.text` section with entropy ~ 0.74. This makes sense since the use of a single instruction type causes more opcode bytes to be identical thus lowering the entropy. However, the entropy difference of the `.text` section may be too neglectable to alone be a good indicator that the file is movfuscated.

However, because the static memory layout contains specific lookup tables, as identified in chapter 6, the file size and full entropy map over sections in the binary will share similarities with other movfuscated programs.

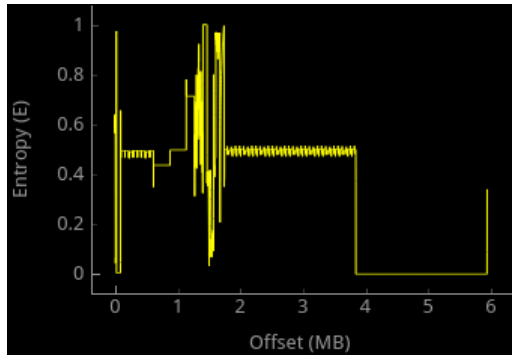


Figure 7.2: Movfuscated entropy of example in listing A.2.

File headers and sections The section names are all standard[Com95] and thus does not reveal that the executable was build by `M/o/Vfuscator2`. However, the sizes of some sections are several megabytes, which can serve as a partial indicator of movfuscation.

```
r@debian:~/Desktop/mal$ readelf -e suspicious | grep "Section Headers" -A 17
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        080480d4 0000d4 000013 00   A  0   0  1
  [ 2] .hash             HASH            080480e8 0000e8 000030 04   A  3   0  4
  [ 3] .dynsym           DYNSYM          08048118 000118 000070 10   A  4   1  4
  [ 4] .dynstr           STRTAB          08048188 000188 000048 00   A  0   0  1
  [ 5] .gnu.version      VERSYM          080481d0 0001d0 00000e 02   A  3   0  2
  [ 6] .gnu.version_r    VERNEED         080481e0 0001e0 000020 00   A  4   1  4
  [ 7] .rel.plt          REL             08048200 000200 000030 08  AI  3   8  4
  [ 8] .plt              PROGBITS        08048230 000230 00007c 04  AX  0   0 16
  [ 9] .text             PROGBITS        080482ac 0002ac 003c59 00  AX  0   0  1
  [10] .eh_frame         PROGBITS        0804bf08 003f08 000000 00   A  0   0  4
  [11] .dynamic          DYNAMIC         0804c000 004000 0000a8 08  WA  4   0  4
  [12] .got.plt          PROGBITS        0804c0a8 0040a8 000024 04  WA  0   0  4
  [13] .data             PROGBITS        0804c0d0 0040d0 5a92ec 00  WA  0   0 16
  [14] .bss              NOBITS          085f53c0 5ad3bc 200010 00  WA  0   0 16
  [15] .shstrtab         STRTAB          00000000 5ad3bc 00007b 00      0   0  1
```

Figure 7.3: The section headers are visible, but does not indicate that the file is movfuscated.

Loaded libraries For this part of the analysis we recompiled the example with `M/o/Vfuscator2` while linking explicitly with the *libcurl* library.

```
r@debian:~/Desktop/mal$ readelf -d suspiciousWithCurl | grep -i shared
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x00000001 (NEEDED)                     Shared library: [libm.so.6]
 0x00000001 (NEEDED)                     Shared library: [libcurl.so.4]
```

Figure 7.4: Imported libraries are not obfuscated.

7.3 Basic Dynamic Analysis

In this section we will focus on basic dynamic analysis of the example program in listing A.2 compiled by the `M/o/Vfuscator2`. The goal is to uncover which type information can be derived from executing the movfuscated example program in a lab environment.

File activity Seeing which files a program opens, reads, and writes to can reveal important aspects of its functionality. In this case we can, for instance, see that the program opens a file named "log" and writes "Log: executing[...]" to it.

```
r@debian:~/Desktop/mal$ strace -e open,write -f ./suspicious 2>&1|grep -i log
open("log", O_RDWR|O_CREAT|O_APPEND, 0666) = 3
write(3, "Log: executing nc -w 1 -e /bin/s"..., 156) = 156
```

Figure 7.5: Using strace it is revealed that the example program writes to a log file.

Registry activity This is mostly relevant for operating systems that uses a registry such as versions of Microsoft Windows. Since we analyse an Linux ELF binary this category is irrelevant.

Network activity Using a network monitoring tool, such as tcpdump[1] or Wireshark[2], to capture network packets while executing the example program reveals that it tries to connect to three different IP-addresses.

No.	Destination	Protocol	Src port	Dst port	Info
1	192.168.100.1	TCP	52495	1337	52495→1337 [SYN] Seq=0 Win=29200 Len=
2	192.168.100.1	TCP	52495	1337	[TCP Retransmission] 52495→1337 [SYN]
3	192.168.100.2	TCP	35084	1337	35084→1337 [SYN] Seq=0 Win=29200 Len=
4	192.168.100.2	TCP	35084	1337	[TCP Retransmission] 35084→1337 [SYN]
5	192.168.100.3	TCP	60802	1337	60802→1337 [SYN] Seq=0 Win=29200 Len=
8	192.168.100.3	TCP	60802	1337	[TCP Retransmission] 60802→1337 [SYN]

Figure 7.6: Dynamic analysis reveals attempt of network communication.

Process activity Process activities are visible, including that the process forks and calls execve several times.

[1] http://www.tcpdump.org/
[2] https://www.wireshark.org/

```
r@debian:~/Desktop/mal$ strace -c -f ./suspicious 2>&1 | tail -n 20
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         2           read
  0.00    0.000000           0         1           write
  0.00    0.000000           0         4           open
  0.00    0.000000           0         4           close
  0.00    0.000000           0         3           waitpid
  0.00    0.000000           0         3           execve
  0.00    0.000000           0         4         4 access
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         2           munmap
  0.00    0.000000           0         4           clone
  0.00    0.000000           0         3           mprotect
  0.00    0.000000           0         3           nanosleep
  0.00    0.000000           0        23           rt_sigaction
  0.00    0.000000           0        15           rt_sigprocmask
  0.00    0.000000           0        10           mmap2
  0.00    0.000000           0         4           fstat64
  0.00    0.000000           0         1           set_thread_area
------ ----------- ----------- --------- --------- ----------------
```

Figure 7.7: Dynamic analysis reveals the process activity.

Since `M/o/Vfuscator2` in itself applies no anti dynamic analysis techniques, this phase can potentially reveal the functionality of a movfuscated program with very little effort. However, a program could still use additional techniques to alter it's behavior if it detects attempts at analysis. Therefore an analyst can not be certain of a program's true functionality without applying advanced static analysis.

7.4 Advanced Static Analysis

By disassembling the program it becomes clear that we are dealing with a movfuscated file. While a `MOV` instruction is the most common x86 instruction[Kan], this program contains almost exclusively *MOV* instructions. The small example program A.2, which consist of 28 lines of C-code, has been movfuscated into a program consisting of 3400+ *MOV* instructions. In comparison, compiling the same source code with GCC produces a `.text` section consisting of only 220 instructions.

The interactive disassembler (IDA[3]) is unable to properly identify function boundaries of the movfuscated executable as illustrated in figure 7.8.

[3]For this analysis the trial version of IDA 6.9.160225 was used.

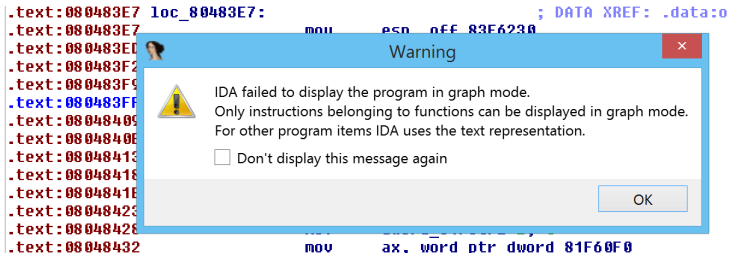


Figure 7.8: IDA is not able to define functions in the movfuscated program, and the graph view is therefore not allowed.

There are several reasons that IDA is having trouble with defining the functions. Traditionally, functions are invoked using `CALL` and function endings are indicated by a *RET* instruction. Individual functions such as *main* and *reverse_shell* in our example program are therefore nowhere to be seen, because as we described in section 6.5.4, `M/o/Vfuscator2` instead uses `execute on` and `execute off`. Secondly the last instruction of the *master loop* is `mov cs, eax` with opcode *0x8ec8*, the illegal instruction to restart the programs instruction listing. IDA does not recognise the opcode for this instruction and the disassembly of it fails, which in turn leads to en error in defining the function. This can be fixed manually by patching out the illegal instruction before defining the function, which allows IDA to show the graph view. However, where one normally would expect to see a control flow graph arrows indicating conditional jumps, loops, and such, IDA shows just a huge block of *MOV* instructions. Figure 7.9 illustrates the size difference between the graph view of the movfuscated file and relevant graph views from an equivalent program compiled with GCC.

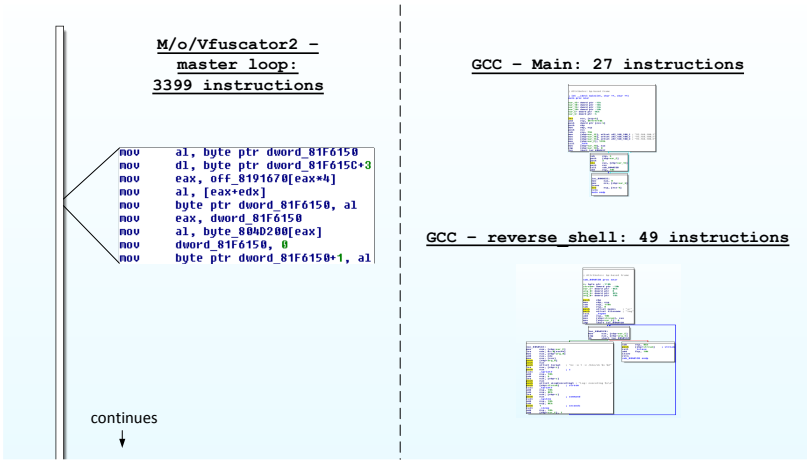


Figure 7.9: Graph view in IDA of the master loop in a movfuscated program in comparison with the graph view of two separate functions from the same program compiled with GCC.

At this point it is clear that the advanced static analysis phase is severely crippled by the lack of detection of control flow combined with the relatively large number of instructions. To conduct a thorough static analysis of the program, we need a method for deobfuscation of a movfuscated program. Section 7.6 will describe one such method.

7.5 Advanced dynamic analysis

Advanced dynamic analysis can be helpful in analysing packed programs. However, dynamic analysis is of little benefit in the scenario of a movfuscated program as there are no transitional execution states where the program is deobfuscated in memory.

7.6 Deobfuscation

The following section details our work on implementing a deobfuscator for `M/o/Vfuscator2`. In particular, the tool makes the output of the movfuscator easier to analyse by reintroducing control flow, call conventions, the stack,

function layout, and reduces the number of instructions.

7.6.1 Overview

Figure 7.10 shows the structure of the deobfuscator. First the deobfuscator accepts a 32-bit elf binary, and the `.text` section is extracted using *libelf*, and then disassembled using *Capstone*[4]. Next the pattern matching module uses a combination of regex matching and `.data` section lookups to match movfuscation primitives. The result of the pattern matching phase is an abstract syntax tree (AST) which contains pattern boundaries and meta information such as jump addresses and source and destination operands. The optimizer then uses the AST to deobfuscate the instruction listing by replacing movfuscation primitives. In order to preserve pattern boundaries instruction sequences that are replaced are `NOP` extended. This results in an instruction listing with many `NOP` instructions between small sequences of actual instructions. The post module relocates `NOP` instructions to after the `RET` instruction. Using this `NOP` relocation strategy means the absolute start position of the functions remains the same so that both function pointers and debug symbols - if present - still works. The relocation of `NOP` instructions causes IDA to effectively ignore the `NOP` instructions. Finally, the deobfuscated instruction listing is assembled into a file.

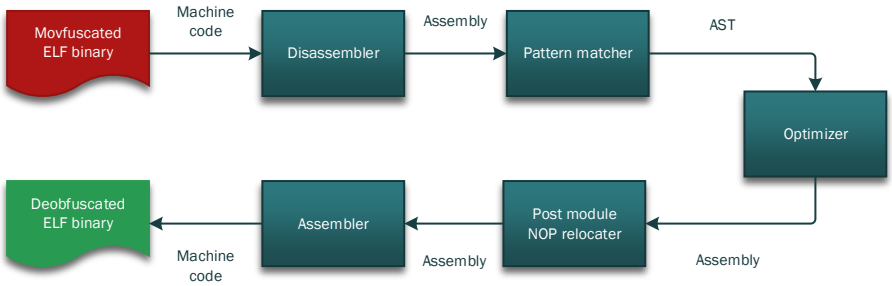


Figure 7.10: The process of deobfuscating a movfuscated ELF binary.

7.6.2 Pattern matcher

Pattern-based deobfuscation is considered a simple and efficient technique that can defeat obfuscated code that use a limited set of patterns[BB14, p. 312]. According to Domas, and in consistency with our analysis detailed in chapter

[4]http://www.capstone-engine.org/

6, the `M/o/Vfuscator2` uses basic building blocks, referred to in this project as movfuscation primitives, to compile code [Dom15a].

The following listing of regular expressions matches `execute on` and `execute off` primitives. The regular expressions in the listing have been developed by examining the source code of `M/o/Vfuscator2`[Domc]. Regular expressions for other primitives have been created in a similar manner.

```
1 mov eax, dword ptr \\[0x.*\\]
2 mov eax, dword ptr \\[eax[*]4 + 0x.*\\]
3 mov dword ptr \\[eax\\], [01]
```

More complex patterns consist of a hierarchy of other primitives. For example figure 7.11 shows an AST of a `function` that has been pattern matched. The function makes an internal call followed by an external call. Before the first internal call an argument is pushed to the stack. Likewise two arguments are pushed before the external call. Matching larger patterns allows us to semantically replace primitives depending on the context they appear in.

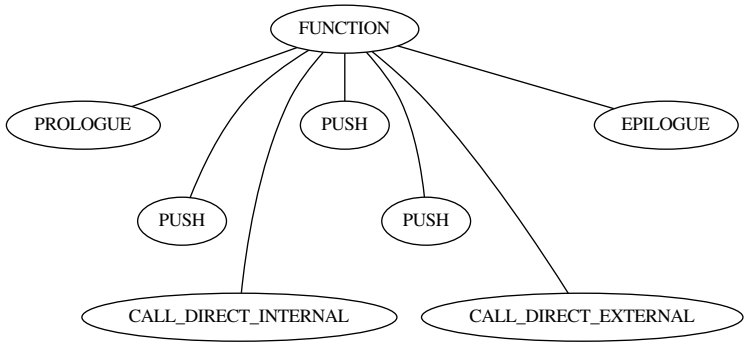


Figure 7.11: Graph of main function of the fib program shown in listing A.4

7.6.3 Reintroducing control flow

Control flow is essential so that tools like IDA Pro can compute and display useful control flow graphs.

In our analysis of `M/o/Vfuscator2` we found that movfuscated programs execute all instructions in chronological order until the last instruction, which causes the program to restart execution. We also found that jumping is done by toggling execution off and later on again once the jump target is reached. In particular, before an `execute off` primitive a jump target is set and instructions are skipped until an `execute on` primitive is reached that successfully matches the jump target. Our decompiler must identify relevant jump primitives.

Furthermore, our analysis revealed that to avoid contamination of *virtual registers* these need to be stored before a jump and restored after. In addition, conditional jumps must first compute the condition while calls have a return target build using an execute off.

Finally, movfuscated calls have a return target so that functions can return execution to the primitive following the call. To complement this, internal functions use a special control flow primitive to return. External movfuscated calls are instead handled by an exception handler.

Table 7.1 list simplified sequence patterns for different types of movfuscated control flow primitives. The actual sequence have a number of glue instructions. There is also code that setup arguments before calls and clean up the stack when control returns.

To reintroduce control flow the matched sequences of building blocks are replaced by our decompiler to regular x86 control flow instructions. Except for return, the jump target address is extracted from the matched sequence and converted to a relative offset. *Unconditional jumps* are replaced by `JMP`, *conditional jumps* are replaced by `JNE, JGE, JG, JLE, JL,` and `JE`, external and internal calls are placed by `CALL`, and *Returns* are replaced by `LEAVE` and `RET`. Since the movfuscated program works on the *virtual stack* the replacement for *return* will not work before the stack has been reintroduced.

7.6.4 Reintroducing the stack

Reintroducing the stack and stack frames for function is essential so that tools like IDA Pro can identify functions, and show local variables and arguments.

In chapter 6 we found that movfuscated programs operate on two stacks: a *virtual stack* and a *virtual discard stack*. Reintroducing the stack requires us to:

1. Rewrite code primitives that pushes or pops from the stack.

Matching control flow

Unconditional jump

1. `Store target`
2. `Store register`
3. `Execute off`

Conditional jump

1. `Jump condition`
2. `Store target`
3. `Store register`
4. `Execute off`

Internal Call

1. `Push return`
2. `Store target`
3. `Store register`
4. `Execute off`
5. `Execute on`

Return

1. `Pop register`
2. `Store target`
3. `Store register`
4. `Execute off`

External Call

1. Push return
2. `Signal SIGSEGV`.

Table 7.1: Sequences of primitives that match different control flow types.

2. Build a new stack frame that follows x86 *cdecl* conventions, described in chapter 5.
3. Rewrite code primitives that access local variables and arguments.

1) Is simply a matter of matching the *virtual stack* push and pop `MOV` sequences emitted by `M/o/Vfuscator2` and replacing them with `PUSH` and `POP` respectively.

2) The movfuscated prologue of a function sets up a virtual stack frame but does not follow the conventions described in chapter 5. Reintroducing the local stack frame consist of replacing the movfuscated prologue in two steps. The first step is to setup the frame pointer (`EBP`) by pushing `EBP` to the stack and then copying the stack pointer (`ESP`) to `EBP`. In the second step, the size of the local variables are computed from the movfuscated code and then subtracted from `ESP`.

```
1 push ebp
2 mov esp, ebp
3 sub esp, size
```

Since the stack frame is now reintroduced, a function can return as shown below. This gives IDA ideal conditions for detecting functions boundaries:

```
1 leave
2 ret
```

3) As a result of rewriting the *stack frame*, the address offsets that were used for the *virtual stack frame*, to access local arguments and variables, have become invalid. To fix this, primitives used to modify and retrieve variables on the *virtual stack* are identified and replaced with new primitives that relies on `EBP` and has adjusted offsets.

7.6.5 Instruction reduction

Instruction reduction reduces the number of instructions that an analyst must consider in order to understand the meaning of the program. While the reintroduction of control flow and the stack in itself reduces the number of instructions significantly, primitives that are not part of replaced patterns such as arithmetic operations still occupy many instructions each. For example, a movfuscated `ADD` primitive consists of 24 instructions, but can be reduced to three or less.

7.6.6 Summary

The result of our deobfuscation process is a significant reduction in size and complexity of a program compiled with `M/o/Vfuscator2`. Since no symbol

information is used, stripping the file has no our deobfuscation.

For example the fib program in listing A.4 have been reduced from 2436 instruction to 132. In fact the instruction count and control flow resembles what Gnu Compiler Collection (GCC) produces as shown in figure 7.12.

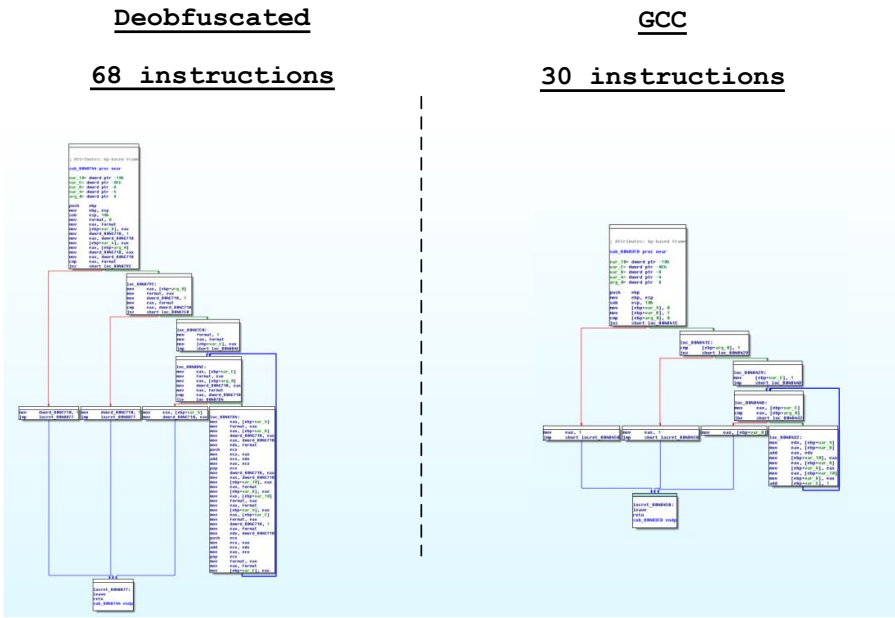
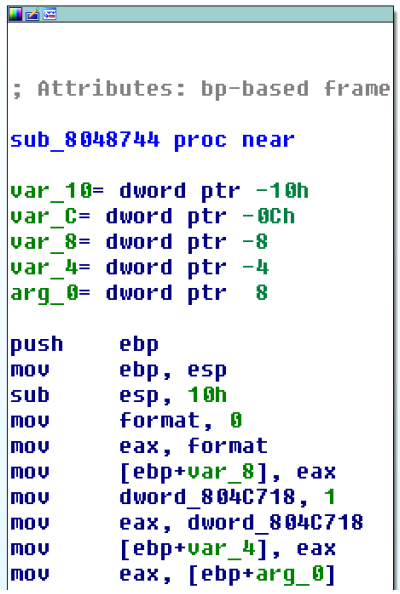


Figure 7.12: Graph of the fib function of the Fibonacci program shown in listing A.4

Functions and variables are now recognized by IDA as shown in figure 7.13.



```
; Attributes: bp-based frame

sub_8048744 proc near

var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 10h
mov     format, 0
mov     eax, format
mov     [ebp+var_8], eax
mov     dword_804C718, 1
mov     eax, dword_804C718
mov     [ebp+var_4], eax
mov     eax, [ebp+arg_0]
```

Figure 7.13: Shows how IDA can detect the local function and variables of the fib function of the program shown in listing A.4

Calls are recognized properly which makes analysis such as function graphs possible as shown in figure 7.14.

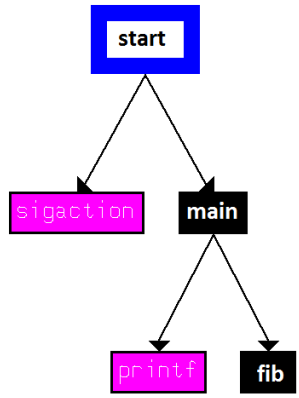


Figure 7.14: Shows that IDA can make a call graph for listing A.4. Note that the visibility of labels have been enhanced post screen shot.

7.6.7 Limitations

The deobfuscator discussed in this report is still a proof of concept and only a subset of possible C constructs have been implemented. Testing have not been done rigidly even for supported operations. However, most of the limitations are straight forward to implement now that the major challenges such as control flow and the stack have been resolved. Known limitations include:

- Non double word sized local variables and arguments.
- Indirect calls e.g. function pointers.
- Only the arithmetic operations `ADD` and `SUB` are reduced in instruction count.
- Obfuscated programs still rely on the *Virtual registers*. The use of these, however, are not far from the use of temporary variables often used in more conventional programs.

In addition to the these limitations, our solution does not make an effort to defeat the post-process modules proposed by Domas[Domb]. However, the work by Kirsch and Jonischkeit [KJ16][Jon16] have shown that taint analysis can be used against movfuscated code hardened by the post-process modules.

Obfuscation Techniques

The field of obfuscation offers a vast amount of techniques to counter advanced static analysis of assembly. While obfuscation can be used on it's own, it is also used in packed malware[RM13][SH12, p. 384]. Packed malware must have a runable unpacking stub that can unpack it's payload, and obfuscation can protect this stub from analysis[SH12, p. 384].

We focus on a selection of techniques described by Dang et al[BB14] that we find relevant in the context of `M/o/Vfuscator2`. The techniques described in this chapter can be divided into three categories: *data based obfuscation*, *control based obfuscation*, and *simultaneously control-flow and data-flow obfuscation*[BB14, chp. 5], as illustrated in figure 8.1.

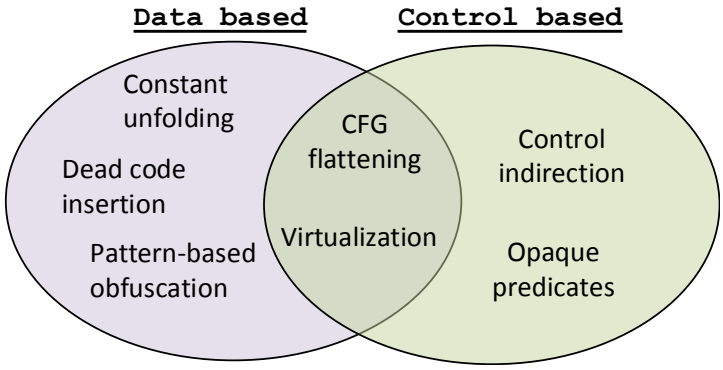


Figure 8.1: Overview of the obfuscation techniques discussed in this chapter.

8.1 Data Obfuscation

Dang et al. defines *data based obfuscation* as obfuscation that increases the complexity without altering the control flow[BB14, p. 273]. Many of these techniques are the reverse of common program optimization techniques.

8.1.0.1 Constant unfolding

This technique makes deciphering of constant values at a glance more difficult. Instead of directly using constants, a set of instructions is inserted which compute the constant at runtime.

Consider the scenario in listing 8.1. The code is about to execute a system command. The argument to the function is located at `EBP+0x8`, but instead of just using `push DWORD PTR [ebp+0x8]` to push the argument the constant 0x8 is unfolded to the three instructions.

```
1 mov     edx,0x2                ; edx = 2
2 inc     edx                    ; edx = 3
3 add     edx,0x5                ; edx = 8
4 push    DWORD PTR [ebp+edx]    ; edx = 8
5 call    80482c0 <system@plt>
```

Listing 8.1: An example of constant unfolding.

8.1.0.2 Pattern-based

This technique matches instructions of one pattern and replaces them with a more complex pattern with the same computation effect as illustrated by listing 8.2. This technique can be applied iteratively in order to increase obscurity.

```
1 xor  eax, edx
2 xor  edx, eax
3 xor  eax, edx
```

Listing 8.2: An example of how `xchg eax, edx` can be replaced with a new pattern.

Finding an equivalent pattern can be difficult if there are requirements to also preserve the computation effect of, e.g., flag registers. This can make it difficult

to apply this technique on compiled code[BB14, p. 277]. This technique can be defeated by writing inverse patterns substitutions or by applying peephole optimization[BB14, p. 277].

8.1.0.3 Dead code insertion

This technique aims to clutter the instruction listing with instructions that does not change the computational effect of the program.

The first two assignments to `EAX` in listing 8.3 are dead since they have not effect on the programs outcome. Normally a liveness analysis removes such instances but this obfuscation technique instead uses them to increase complexity.

```
1 xor     eax,ds:0xffffffeca    ; dead
2 add     eax,0x4               ; dead
3 mov     eax, 0x1              ; live
4 push    eax
5 call    0x80482c0
```

Listing 8.3: An example of dead code insetion.

8.2 Control Obfuscation

Control based obfuscation exploits, that compilers use control flow constructs in a standard and predictable way[BB14, p. 278]. For instance, a compiler will use the call instruction to call other functions and later return and continue execution from where the `CALL` instruction was encountered. However, the instruction can also be used as a jump that puts the address of the next byte on the stack. The same goes for conditional jumps which can be used in unconventional ways to make the analysis more difficult and time consuming.

8.2.0.1 Control indirection

This technique uses signal or exception handlers to set up jumps[SH12, p. 344-346]. For example, a signal handler specifies a target address to which to jump[pro]. The exception handler can then be triggered by causing an error that sends the specified signal, e.g `SIGSEGV` is invoked when an instruction tries to access memory outside a readable segment.

Exception triggered control flow can cause reverse engineering tools that rely on detection of conventional control flow mechanisms to fail to detect jumps masked as exception handlers [SH12, p. 344].

8.2.0.2 Opaque predicates

This technique relies on jumps that appear conditional but are in reality predetermined i.e. opaque. Such jumps can be used to make the control flow appear more complex and can be written so that it is difficult to determine statically whether the jump is taken or not [BB14, p. 283]. Figure 8.2 illustrates how an opaque predicate can increase the complexity of the control flow graph.

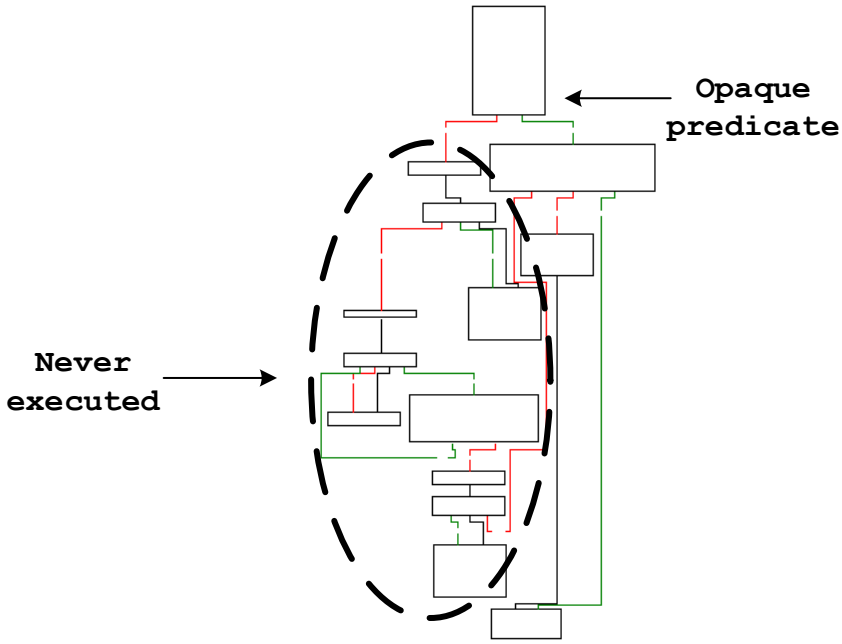


Figure 8.2: An example of a opaque predicate shown in IDA, which always evaluate to true making the left path essentially junk code.

8.3 Combined Obfuscation

The last category of obfuscation techniques are targeting both the control- and data-flow.

8.3.1 Control-flow graph flattening

This technique flattens the control flow graph by making control flow operations go through a common dispatcher. Each dispatched piece of code is responsible for indicating which block of code should be called next[BB14, p. 285].

Figure 8.3 shows the dispatcher `fib` from listing A.5. The `fib` function have been split into multiple functions. The *CALL* instruction in the left most block calls `EAX` which contains the address of the next function. Each called function updates `EAX` to set the next function index before returning. The address `off_8049844` contains a table of functions.

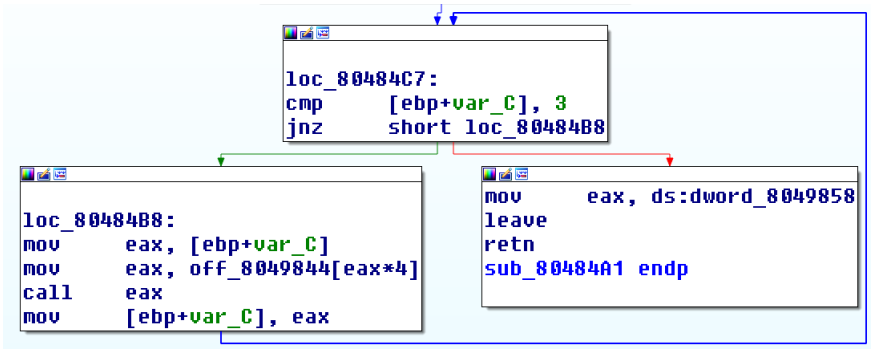


Figure 8.3: A screenshot of a dispatcher from IDA. Each called code fragment function select the next piece of code by returning an id in `EAX`.

8.3.2 Virtual machine based obfuscaton

This technique stores the program as byte code. The byte code is interpreted by a virtual machine that has a unique architecture. In order to analyse the byte code, the analyst has to decipher how the virtual machine executes the byte code, i.e. reverse engineer which instructions are supported by the virtual machine, and how they are encoded[BB14, p. 286].

Standard tools will only analyse and graph the VM code but not the bytecode which is executed by the VM[Rol09]. So to understand the bytecode the analyst will often need to create a compiler from the VM's custom architecture to a traditional architecture such as *x86*.

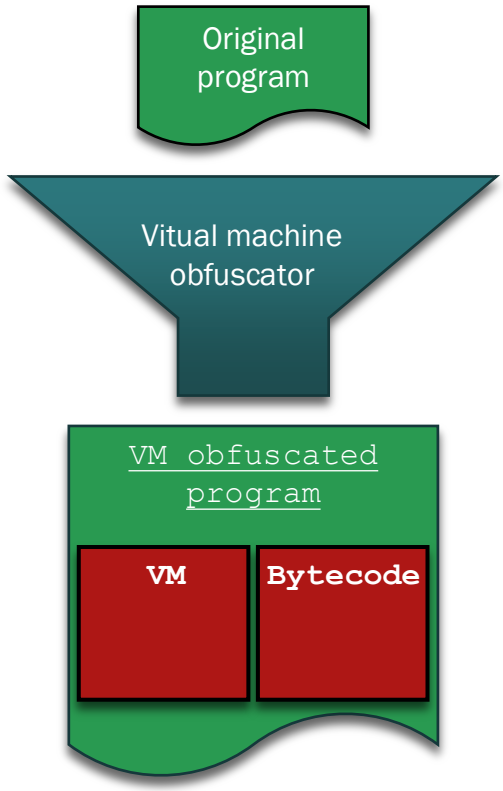


Figure 8.4: Overview of a virtual machine based obfuscator.

Evaluation

To evaluate single instruction compiling as an obfuscation technique, we first classify `M/o/Vfuscator2` relative to the techniques described in chapter 8. This will allow us to draw conclusions on the effectiveness of `M/o/Vfuscator2` based on the effectiveness of the techniques that it shares traits with.

Since `M/o/Vfuscator2` is an implementation of single instruction compiling, we will then use it to draw conclusions on the effectiveness on single instruction compiling as an obfuscation technique. However, since an implementation does not necessarily reflect the effectiveness of single instruction obfuscation in general, we will attempt to abstract from implementation limitations.

From our analysis we know that `M/o/Vfuscator2` is compiled using basic building blocks to increase code complexity. This shares simularity with pattern based obfuscation. But unlike pattern based obfuscation `M/o/Vfuscator2` also targets control flow.

`M/o/Vfuscator2` also shares a similarity with dead code insertion. However, while dead code can be removed without changing the computational effect of the program - this is not the case for instructions in a movfuscated program.

Control indirection is employed by `M/o/Vfuscator2` to handle both external calls and to restart the master loop.

In contrast to opaque predicates which increase the complexity of the control flow, programs compiled by `M/o/Vfuscator2` essentially eliminates it.

The approach of `M/o/Vfuscator2` is more similar to that of control-flow graph

flattening which aims to prevent the leakage of control flow information by using a common dispatcher. A similar principle is used by movfuscated code. In particular, instead of using jumps, code blocks update a global branch target before toggling execution off. By skipping instructions the target code block is eventually reached and execution is turned on again. This can be thought of as an inlined dispatcher.

Of all the techniques `M/o/Vfuscator2` may have most in common with virtual machine based obfuscation. Both techniques employ a combination of data and control based obfuscation. They also both requires the analyst to decipher the architecture and then build a deobfuscator. In effect it a movfuscated program can be thought of as a virtual machine with a fixed guest program. This contrasts to a stored program computer, which can run any program stored in memory.

The primary strength of conventional virtual machine based obfusaction lies in the difficulty of analysing how the VM interprets bytecode. This process can be very time consuming[BB14, p. 286].

However, a weakness of `M/o/Vfuscator2` is that the architecture is fixed unlike state of the art VM obfuscators, such as VMProtect[1], that creates a custom instruction set for each program that is obfuscated. This is highlighted by the fact that after having received academic attention, two deobfuscators - one developed by us and one that is already publicly available[KJ16] - have been developed.

In essence, `M/o/Vfuscator2`, like VM obfuscation, provides security by obscurity but while state of the art VM obfuscation can apply polymorphism to obscure each sample uniquely[AJV06], `M/o/Vfuscator2` only has to be broken once.

Dolan and Domas worked with several self imposed restrictions that need not affect single instruction compiling in general: the single instruction should be an x86 instruction, code modification was disallowed, and memory mapped computational units, as seen in transport triggered architectures (TTA), should not be used [Dol13][Dom15a][Dom15b][Dom15c].

If we remove these restrictions we can design a virtual machine with a one instruction set computing (OISC) architecture, that interprets a custom instruction that may not be natively be supported on x86.

A TTA can emulate a RISC instruction set by mapping triggered memory addresses with the semantics of the different RISC instructions. Because of

[1] http://vmpsoft.com/

this, state of the art VM obfuscators that are already able to generate unique RISC[Rol09] instruction sets for each protected file could instead work by generating unique OISC architectures. This effectively means that OISC based VM obfuscation could be as effective and diverse as RISC based VM obfuscation. This holds under the assumption that reverse engineering a VM with a single instruction TTA is as hard as reverse engineering a VM with a RISC architecture. In addition, while several approaches to defeat VM obfuscators have been proposed[Rol09][Sha+09], it is still considered a potent obfuscation technique [BB14, p. 286].

Our analysis shows that single instruction compiling is effective against advanced static analysis. This is of most relevancy when a malware sample also employs anti analysis techniques against basic analysis, as the lack of such techniques may allow basic analysis to uncover the functionality of the sample faster thus rendering advanced static analysis unnecessary. As soon as the sample has been analysed actions can be taken to counter the malware attack as described by the *Cyber Kill Chain*[Eri11]. Thus single instruction compiling is most effective when combined with other obfuscation techniques that counter basic analysis.

Conclusion

The objective of this report was to evaluate single instruction compiling as an obfuscation technique by we found that `M/o/Vfuscator2` is similar to virtual machine obfuscation. Virtual machine based obfuscation is seen as a potent obfuscation techniques against advanced static analysis.

`M/o/Vfuscator2`, however, only has a single way to interpret and encode instructions, once the encoding has been analysed it becomes less valuable as an obfuscation technique. This is unlike state of the art virtual machine obfuscators that generate unique instruction sets for each protected program. This is a limitation of `M/o/Vfuscator2`, and not single instruction computing in general. Thus virtual machine obfuscation may leverage single instruction compiling just as it leverages any other custom instruction set compiler.

10.1 Future Work

Formally verify that reverse engineering a VM with a single instruction set architectures is as hard as reverse engineering as a VM with a RISC architecture.

Further research the possibility of implementing VM based obfuscators that leverages a custom single instruction set architecture for each protected file.

Example programs

A.1 Programs in C

```c
#include <stdio.h>

int main(void)
{
  printf("Hello, world!\n");
  return 0;
}
```

Listing A.1: C-Program 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

void reverse_shell(const char **ip_list, size_t
    ip_count, unsigned port) {
  char cmd[256];
  int i;

  for(i = 0; i<ip_count; ++i){
    sprintf(cmd, "nc -w 1 -e /bin/sh %s %d", ip_list[i
       ], port);
```

```
11      system(cmd);
12      sleep(1);
13    }
14  }
15
16  int main() {
17    const char *ip_list[] = {"192.168.100.1","
          192.168.100.2","192.168.100.3"};
18    unsigned port = 1337;
19    pid_t child = fork();
20    if(child == 0) {
21      reverse_shell(ip_list, 3, port);
22    }
23    return 0;
24  }
```

Listing A.2: C-Program: Example program that forks and them attempts to establish reverse shells to three different IP adresses.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/types.h>
4
5   void reverse_shell(const char **ip_list, size_t
        ip_count, unsigned port) {
6     char cmd[256];
7     int i;
8     FILE *fp;
9     fp = fopen("log", "a+");
10
11    for(i = 0; i<ip_count; ++i){
12      sprintf(cmd, "nc -w 1 -e /bin/sh %s %d", ip_list[i
            ], port);
13      fprintf(fp,"Log: executing %s\n", cmd);
14      system(cmd);
15      sleep(1);
16    }
17    fclose(fp);
18  }
19
20  int main(char **argv, int argc) {
21    const char *ip_list[] = {"92.168.100.1","
```

```
      92.168.100.2","92.168.100.3"};
22   unsigned port = 1337;
23   pid_t child = fork();
24   if(child == 0) {
25     reverse_shell(ip_list, 3, port);
26   }
27   return 0;
28 }
```

Listing A.3: C-Program: Example program that forks and them attempts to establish reverse shells to three different IP adresses. It also logs each attempt to a local file.

```
1  #include <stdio.h>
2  #include <stddef.h>
3  #include <stdlib.h>
4  #include <ctype.h>
5
6  unsigned int fib(unsigned int n)
7  {
8     unsigned int i, j, k, t;
9     i=0; j=1;
10
11    if(n==0){
12   return 1;
13    }else if(n==1){
14   return 1;
15    }
16
17    for (k = 1; k <= n; ++k)
18    {
19      t = i + j;
20      i = j;
21      j = t;
22    }
23    return j;
24  }
25
26 int main(int argc, char** argv){
27   printf("%d\n", fib(10));
28 }
```

Listing A.4: C-Program: Example program that computes fibonacci numbers.

```
1
2  #include <stdio.h>
3  #include <stddef.h>
4  #include <stdlib.h>
5  #include <ctype.h>
6
7  #define SETUP 0
8  #define CHECK 1
9  #define LOOP 2
10 #define DONE 3
11
12 unsigned int n, i, j, k, t;
13
14 int  setup() {
15     i=0; j=1; k=1;
16     return CHECK;
17 }
18
19  int check(){
20    if(n==0 || n == 1){
21   return DONE;
22    }
23    return LOOP;
24 }
25
26  int loop(){
27    if (k <= n){
28      t = i + j; i = j; j = t; k++;
29      return LOOP;
30    }
31    return DONE;
32 }
33
34 int (*funcs[])(void) = {setup, check, loop};
35
36 unsigned int fib(unsigned int _n){
37    n=_n;
38    int state=SETUP;
```

```
39    while(state != DONE){
40     state = funcs[state]();
41    }
42    return j;
43 }
44
45 int main(int argc, char** argv){
46   printf("%d\n", fib(10));
47 }
```

Listing A.5: C-Program: Fib program that is flattened to obfuscate control flow

Bibliography

[Cyb16] Center For Cybersikkerhed. *Cybertruslen mod Danmark*. 2016. URL: https://fe-ddis.dk/cfcs/CFCSDocuments/Cybertruslen%5C%20mod%5C%20Danmark%5C%202016.pdf (visited on 03/11/2016).

[Sec15] Kaspersky Security. *KASPERSKY SECURITY. BULLETIN 2015*. 2015. URL: https://securelist.com/files/2015/12/Kaspersky-Security-Bulletin-2015_FINAL_EN.pdf (visited on 03/11/2016).

[Ver15] Verizon. *2015 Data Breach. Investigation report*. 2015. URL: https://securelist.com/files/2015/12/Kaspersky-Security-Bulletin-2015_FINAL_EN.pdf (visited on 03/11/2016).

[Mica] Microsoft. *Defining Malware: FAQ*. URL: https://technet.microsoft.com/en-us/library/dd632948.aspx (visited on 07/02/2016).

[Bul] Bullguard. *Malware - definition, history and classification*. URL: http://www.bullguard.com/da/bullguard-security-center/pc-security/computer-threats/malware-definition,-history-and-classification.aspx (visited on 07/02/2016).

[Kasa] Kaspersky. *What is Malware and How to Defend Against It?* URL: http://usa.kaspersky.com/internet-security-center/internet-safety/what-is-malware-and-how-to-protect-against-it#.V3ewUu0vD0o (visited on 07/02/2016).

[Sym] Symantec. *What are malware, viruses, Spyware, and cookies, and what differentiates them ?* URL: http://www.symantec.com/connect/articles/what-are-malware-viruses-spyware-and-cookies-and-what-differentiates-them (visited on 07/02/2016).

[Kasb] Kaspersky. *The Epic Turla (snake/Uroburos) attacks*. URL: http://www.kaspersky.com/internet-security-center/threats/epic-turla-snake-malware-attacks (visited on 05/24/2016).

[Lai16] Allison Nixon Laith Alkhouri Alex Kassirer. *Hacking for ISIS: The Emerging Cyber Threat Landscape*. Flashpoint, 2016.

[Dom15a] Christopher Domas. "The M/o/Vfuscator". Derbycon 2015. 2015. URL: http://www.irongeek.com/i.php?page=videos/derbycon5/break-me00-the-movfuscator-turning-mov-into-a-soul-crushing-re-nightmare-christopher-domas.

[Dom15b] Christopher Domas. "The M/o/Vfuscator". Recon 2015. 2015. URL: https://recon.cx/2015/schedule/.

[Dom15c] Christopher Domas. "REpsych: Psychological Warfare in Reverse Engineering". DEFCON 23. 2015. URL: https://www.defcon.org/html/defcon-23/dc-23-speakers.html#Domas.

[Doma] Christopher Domas. *Movfuscator*. URL: https://github.com/xoreaxeaxeax/movfuscator (visited on 07/02/2016).

[Dol13] Steven Dolan. "Mov is Turing-complete". In: (2013).

[Domb] Christopher Domas. *Movfuscator*. URL: https://github.com/xoreaxeaxeax/movfuscator/tree/master/post (visited on 07/03/2016).

[KJ16] Julian Kirsch and Clemens Jonischkeit. "Movfuscator-Be-Gone". Recon 2016. 2016. URL: https://recon.cx/2016/talks/%5C%22Movfuscator-Be-Gone.html.

[Jon16] Clemens Jonischkeit. "Machine Code Obfuscation via Instruction Set Reduction and Control Flow Graph Linearization: Analysis and Countermeasures". Bachelor's Thesis. Technische Universität München, Mar. 2016.

[SH12] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st. San Francisco, CA, USA: No Starch Press, 2012. ISBN: 978-1593272906.

[Mur10] Liam O Murchu. *Stuxnet Using Three Additional Zero-Day Vulnerabilities. Symantec Official Blog*. 2010. URL: http://www.symantec.com/connect/blogs/stuxnet-using-three-additional-zero-day-vulnerabilities (visited on 03/11/2016).

[KM14] Jan-Frederik Kremer and Benedikt Müller. *Cyberspace and International Relations: Theory, Prospects and Challenges*. 1st. Springer, 2014. ISBN: 978-3-642-37480-7.

[Man13] Mandiant. *APT1. Exposing One of China's Cyber Espionage Units*. 2013. URL: http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf (visited on 03/11/2016).

[Gol] David Goldman. *Anonymous attacks Greek Central Bank and vows to take down more banks' sites.* URL: http://money.cnn.com/2016/05/04/technology/anonymous-greek-central-bank/ (visited on 06/30/2016).

[SAN] GENEVA SANDS. *What to Know About the Worldwide Hacker Group 'Anonymous'.* URL: http://abcnews.go.com/US/worldwide-hacker-group-anonymous/story?id=37761302 (visited on 06/30/2016).

[Pau] Darren Pauli. *Anonymous whales on Denmark, Iceland with Op-KillingBay DDoS.* URL: http://www.theregister.co.uk/2016/04/22/anonymous_whales_on_denmark_faroe_islands_with_opkillingbay_ddos/ (visited on 06/30/2016).

[SS14] John Scott-Railton and in collaboration with Cyber Arabs Seth Hardy. *Malware Attacks Targeting Syrian ISIS Critics.* The Citizen Lab, University Of Toronto, 2014.

[Eri11] Rohan M. Amin Eric M. Hutchins Michael J. Cloppert. *Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains.* Lockheed Martin Corporation, 2011.

[Mar15a] Lockheed Martin. *GAINING THE ADVANTAGE: Applying Cyber Kill Chain(C) Methodology to Network Defense.* Lockheed Martin Corporation, 2015.

[Sec12] Dell SecureWorks. *Advanced Threat Protection with Dell SecureWorks Security Services.* Dell SecureWorks, 2012.

[Sig15] Lasse Herløw og Sigurd Jervelund Hansen. "Detection and Prevention of Advanced Persistent Threats". Masters's Thesis. Technical University of Denmark, June 2015.

[Mar15b] Lockheed Martin. *Seven Ways to Apply the Cyber Kill Chain(C) with a Threat Intelligence Platform.* Lockheed Martin Corporation, 2015.

[J V15] J. Vukalovic and D. Delija. "Advanced Persistent Threats - detection and defense". In: *38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015, Opatija, Croatia, May 25-29, 2015.* 2015, pp. 1324–1330. DOI: 10.1109/MIPRO.2015.7160480. URL: http://dx.doi.org/10.1109/MIPRO.2015.7160480.

[DeC] Jessica DeCianno. *Indicators of Attack vs. Indicators of Compromise.* URL: https://www.crowdstrike.com/blog/indicators-attack-vs-indicators-compromise/ (visited on 07/01/2016).

[Kra11] Patrick Kral. *The Incident Handlers Handbook. Investigation report.* 2011. URL: https://www.sans.org/reading-room/whitepapers/incident/incident-handlers-handbook-33901 (visited on 04/01/2016).

[Ley15] John Leyden. *China cuffs hackers at US request to stave off sanctions*. 2015. URL: http://www.theregister.co.uk/2015/10/09/china_cuffs_hackers_at_us_request/ (visited on 04/05/2016).

[Fir13] FireEye. *DIGITAL BREAD CRUMBS. Seven Clues To Identifying Who's Behind Advanced Cyber Attacks*. 2013. URL: https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/rpt-digital-bread-crumbs.pdf (visited on 04/05/2016).

[Nov16] Novetta. *Operation Blockbuster. Unraveling the Long Thread of the Sony Attack*. 2016. URL: https://www.operationblockbuster.com/wp-content/uploads/2016/02/Operation-Blockbuster-Report.pdf (visited on 03/11/2016).

[Fir14] FireEye. *APT28. A WINDOW INTO RUSSIA'S CYBER ESPIONAGE OPERATIONS?* 2014. URL: https://www2.fireeye.com/rs/fireye/images/rpt-apt28.pdf (visited on 04/05/2016).

[Mim16] Michael Mimoso. *APT Attackers Flying More False Flags Than Ever*. 2016. URL: https://threatpost.com/apt-attackers-flying-more-false-flags-than-ever/116814/ (visited on 04/05/2016).

[MR11] Blake Hartstein Michael Hale Ligh Steven Adair and Matthew Ricards. *Malware Analyst's Cookbok and DVD. Tools and techniques for fighting malicious code*. Wiley Publishing, Inc, 2011. ISBN: 9780470613030.

[Micb] Trend Micro. *Ransomware*. URL: http://www.trendmicro.com/vinfo/us/security/definition/Ransomware (visited on 07/02/2016).

[Inv] Federal Bureau of Investigation(FBI). *Protect Your Computer Don't Be Scared by 'Scareware'*. URL: https://www.fbi.gov/news/stories/2010/july/scareware/scareware (visited on 07/02/2016).

[Dic] Oxford Dictionaries. *Obfuscate*. URL: http://www.oxforddictionaries.com/definition/english/obfuscate (visited on 07/02/2016).

[Bar+01] Boaz Barak et al. "On the (Im)possibility of Obfuscating Programs". In: *Advances in Cryptology — CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–18. ISBN: 978-3-540-44647-7. DOI: 10.1007/3-540-44647-8_1. URL: http://dx.doi.org/10.1007/3-540-44647-8_1.

[BB14] Alexandre Gazet Bruce Dang and Elias Bachaalany. *Practical Reverse Engineering*. 1st. Wiley, 2014. ISBN: 1118787315.

[Zel16] Lenny Zeltser. *Mastering 4 Stages of Malware Analysis*. 2016. URL: https://zeltser.com/mastering-4-stages-of-malware-analysis/ (visited on 05/16/2016).

[Car11] H. Carvey. *Windows Registry Forensics: Advanced Digital Forensic Analysis of the Windows Registry*. Windows Registry Forensics: Advanced Digital Forensic Analysis of the Windows Registry. Syngress, 2011. ISBN: 9781597495806. URL: https://books.google.dk/books?id=QwXFlAEACAAJ.

[GL16] Josh Grunzweig and Brandon Levene. *PowerSniff Malware Used in Macro-based Attacks*. 2016. URL: http://researchcenter.paloaltonetworks.com/2016/03/powersniff-malware-used-in-macro-based-attacks/ (visited on 05/16/2016).

[Rap] Rapid7. *POLYMORPHIC XOR ADDITIVE FEEDBACK ENCODER*. URL: https://www.rapid7.com/db/modules/encoder/x86/shikata_ga_nai (visited on 07/02/2016).

[Ins] SANS Institute. *SANS Institute InfoSec Reading Room*. URL: https://www.sans.org/reading-room/whitepapers/forensics/detecting-malware-sandbox-evasion-techniques-36667 (visited on 07/02/2016).

[Int16] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual. Vol. 2. Instruction set reference, A-Z*. Intel, 2016.

[Fog04] Agner Fog. *Calling conventions for different C++ compilers and operating systems*. Technical University of Denmark, 2004.

[Com95] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. TIS Committee, 1995.

[NWH04] Peter J. Nurnberg2004, Uffe K. Wiil, and David L. Hicks. "A Grand Unified Theory for Structural Computing". In: *Metainformatics: International Symposium, MIS 2003, Graz, Austria, September 17-20, 2003. Revised Papers*. Ed. by David L. Hicks. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–16. ISBN: 978-3-540-24647-3. DOI: 10.1007/978-3-540-24647-3_1. URL: http://dx.doi.org/10.1007/978-3-540-24647-3_1.

[MP88] Farhad Mavaddat and Behrooz Parhami. "URISC: The ultimate reduced instruction set computer". In: *International Journal of Electrical Engineering Education* 25.3 (1988), pp. 342–351.

[Jon88] Douglas W. Jones. "The Ultimate RISC". In: *ACM SIGARCH Computer Architecture News* 16.3 (Feb. 1988), pp. 48–55.

[Int08] Maxim Integrated. *MAXQ FAMILY USER'S GUIDE, rev 6*. Maxim Integrated, 2008.

[FH03] C. Fraser and D. Hanson. *The lcc 4.x Code-Generation Interface*. MSR-TR-2001-64. Microsoft, 2003.

[Mog11] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Springer, 2011. ISBN: 978-0-85729-828-7.

[Domc] Christopher Domas. *Movfuscator*. URL: https://github.com/xoreaxeaxeax/movfuscator/blob/master/movfuscator/movfuscator.c (visited on 07/04/2016).

[Domd] Christopher Domas. *Movfuscator*. URL: https://github.com/xoreaxeaxeax/movfuscator/blob/master/post/shuffle.py (visited on 07/03/2016).

[Kan] Peter Kankowski. *x86 Machine Code Statistics*. URL: https://www.strchr.com/x86_machine_code_statistics (visited on 07/07/2016).

[RM13] Kevin A. Roundy and Barton P. Miller. "Binary-code Obfuscations in Prevalent Packer Tools". In: *ACM Comput. Surv.* 46.1 (July 2013), 4:1–4:32. ISSN: 0360-0300. DOI: 10.1145/2522968.2522972. URL: http://doi.acm.org/10.1145/2522968.2522972.

[pro] The Linux man-pages project. *SIGACTION(2)*. URL: http://man7.org/linux/man-pages/man2/sigaction.2.html (visited on 07/02/2016).

[Rol09] Rolf Rolles. *Unpacking Virtualization Obfuscators*. 2009.

[AJV06] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. "Proteus: Virtualization for Diversified Tamper-resistance". In: *Proceedings of the ACM Workshop on Digital Rights Management*. DRM '06. Alexandria, Virginia, USA: ACM, 2006, pp. 47–58. ISBN: 1-59593-555-X. DOI: 10.1145/1179509.1179521. URL: http://doi.acm.org/10.1145/1179509.1179521.

[Sha+09] M. Sharif et al. "Automatic Reverse Engineering of Malware Emulators". In: *2009 30th IEEE Symposium on Security and Privacy*. May 2009, pp. 94–109. DOI: 10.1109/SP.2009.27.