jamk

Analysis of Modern Malware

Obfuscation techniques

Mika Luoma-aho

Master's thesis
May 2023
Master's Degree Programme in Information Technology, Cyber Security

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Luoma-aho, Mika

Analysis of Modern Malware – obfuscation techniques

Jyväskylä: JAMK University of Applied Sciences, May 2023, 102 pages.

Master's Degree Programme in Information Technology, Cyber Security (YAMK). Master's thesis.

Permission for open access publication: Yes

Language of publication: English

Abstract

Threat actors use malware to damage individuals and companies. Due to its adaptability, malware is the most prevalent form of cyber-attack. For example, a cyber-attack can range from a simple virus to a complex ransomware operation. In this never-ending cat-and-mouse game, malware authors devise ever-more complex strategies for bypassing system defences, while cyber security professionals, defence system designers, and endpoint protection developers devise improved methods for identifying these new strategies.

The purpose of the thesis was to assist individuals and businesses in comprehending how malware is currently and will be in the future, evading system protection measures and concealing itself from the analysis.

Since obfuscation techniques are one of the important ways to evade and hide from analysis, the theoretical research looked at various obfuscation techniques, the ability of systems to defend against these techniques, and the possibilities of future innovations.

The findings of the thesis emphasised the need to comprehend the implementation methods of existing malware techniques to comprehend the limitations of current security measures as well as the challenges of researching malware code and anticipating its future advancements.

Because malware authors are always one step ahead of security developers, achieving total protection and anticipating new threats is challenging, if not impossible.

Keywords/tags (subjects)

Malware, obfuscation, detection, mitigation, cyber security, cybersecurity

Miscellaneous (Confidential information)

n/a

jamk

Kuvailulehti

Luoma-aho, Mika

Modernien haittaohjelmien analysointi – suojautumistekniikat

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2023, 102 sivua.

Master's Degree Programme in Information Technology, Cyber Security. Opinnäytetyö YAMK.

Verkkojulkaisulupa myönnetty: Kyllä

Julkaisun kieli: Englanti

Tiivistelmä

Uhkatoimijat käyttävät haittaohjelmia yksilöiden ja yritysten vahingoittamiseen. Haittaohjelmat ovat suosituin kyberhyökkäysmuoto monipuolisuutensa ansiosta. Kyberhyökkäys voi vaihdella yksinkertaisesta viruksesta kehittyneeseen kiristysohjelmaan. Haittaohjelmien tekijät kehittävät jatkuvasti kehittyneempiä tekniikoita järjestelmän suojauksien kiertämiseksi ja samalla kyberturva ammattilaiset sekä virustorjuntaohjelmien sovelluskehittäjät kehittävät parempia keinoja havaita nämä uudet lähestymistavat tässä loputtomassa kissa-hiiri-pelissä.

Koska obfuskointitekniikat ovat yksi tärkeimmistä haittaohjelmien käyttämistä keinoista vältellä ja piiloutua analysoinnilta tässä opinnäytetyössä on tutkittu mahdollisimman kattavasti erilaisia obfuskointitekniikoita ja samalla analysoitu järjestelmien mahdollisuuksia suojautua näitä tekniikoita vastaan, sekä arvioitu minkälaisia mahdollisia uusia kehityssuuntia voi olla tulossa. Työn tavoitteena oli auttaa yrityksiä ja yksityishenkilöitä ymmärtämään haittaohjelmien nykyisiä ja tulevia tapoja vältellä järjestelmien suojautumiskeinoja sekä piiloutua analysoinnilta.

Opinnäytetyön tulos korosti, että on tärkeää ymmärtää nykyisten haittaohjelmien käyttämien tekniikoiden toteutustapoja, jotta voidaan ymmärtää nykyisten suojautumiskeinojen vajaavaisuudet sekä haasteet tutkia haittaohjelmien koodia ja vaikeutta ennakoida niiden tulevia kehityssuuntia.

Koska haittaohjelmien tekijät ovat aina askeleen edellä suojauksien kehittäjiä, on erittäin vaikeaa tai jopa mahdotonta saavuttaa täydellistä suojausta ja ennakointia tulevia uhkia varten.

Avainsanat (asiasanat)

Haittaohjelma, obfuskointi, havaitseminen, torjunta, kyberturvallisuus

Muut tiedot (salassa pidettävät liitteet)

n/a

Contents

Figures

Tables

Abbreviations

AV	Anti-virus
BUG	A fault in a program
CIA	Confidentiality, integrity, and availability
DDoS	Distributed Denial of Service (attack)
EDR	End-point Detection and Response
HIPS	Host Intrusion Prevention System
IDS	Intrusion Detection
IDP	Intrusion Detection and Prevention
IoT	Internet of Things
MBR	Master Boot Record
SMB	Server Message Protocol
SMC	Self-modifying Code
STO	Security Through Obscurity
VM	Virtual Machine
0-day	A vulnerability that is not currently publicly known (also known as Zero-day)

1 Introduction

> *"In an age of dynamic malware obfuscation through operations such as mutating hash, a hyper-evolving threat landscape, and technologically next-generation adversaries, offensive campaigns have an overwhelming advantage over defensive strategies." (James Scott, Senior Fellow, Institute for Critical Infrastructure Technology)*

This thesis was motivated by the escalating threat environment comprising malware as the most severe security threat on the Internet today. Threat actors commonly use malware as their preferred cyber security weapon to target individuals and companies in attack campaigns. Therefore, conducting a comprehensive technical analysis is crucial to gain insights into malware's origins and future trajectory. The research utilized a qualitative research method with an integrative literature review. The research delved into the complex concepts of modern, advanced malware, focusing on how it employs obfuscation as an evasion strategy to avoid detection by various defensive measures used on PCs and cloud-based services.

Malware, an umbrella term for various types of malicious software, applies to any software or code that performs unauthorised operations and compromises the Confidentiality, Integrity, or Availability of a computer system (CIA). Malware, also known as harmful code or logic, can be viruses, worms, trojan horses, and other malicious software that can undermine a targeted system's security and stability (European Union Agency for Cybersecurity, 2022). Malicious software also includes spyware and various forms of adware. A virus is a software replicating itself and infecting other files on a computer, allowing it to spread from host to host. A worm, like a virus, spreads independently over a network, usually causing significant harm. A trojan horse is a malware that disguises itself as legitimate software, deceiving users into installing it. Spyware, on the other hand, covertly tracks and collects personal information without the user's consent. Finally, adware is unwanted software that triggers pop-up ads and banners on a computer (*Malwarebytes*, 2022b).

Malicious actors, throughout their campaigns, frequently utilise malware to obtain and maintain asset control, evade and deceive defenders, and carry out post-compromise operations. Malware authors choose from various malware types to gain control of target systems. The kind of malware they select depends on their specific goals and the vulnerabilities they are trying to exploit. Technically, the malware consists of different components, such as payloads, droppers, post-compromise tools, backdoors, and packers which differ in their functionalities. Attackers determine and

combine the most efficient malware components based on their objectives, from gaining control of systems or networks and stealing data to disrupting operations. (European Union Agency for Cybersecurity, 2022)

Malware authors frequently employ obfuscation as a means of evading antivirus scanners. This technique encompasses a range of tactics, including the use of encrypted, oligomorphic, polymorphic, and metamorphic malware. The primary goal of obfuscation is to make malware code as complex and challenging to comprehend for humans and automated analysis tools as possible. The considerable prevalence of effective obfuscation techniques presents a significant threat to individuals who depend on antivirus software as their principal safeguard against malicious software attacks on the internet. Given the impact of obfuscation on the efficiency of malware detection and analysis, it has become an important research topic in computer security. Utilising advanced analysis techniques and machine learning algorithms, researchers are actively investigating methods to improve the detection of obfuscated malware.

This thesis explores the numerous techniques harmful programmes use to function effectively in the face of more advanced system security measures and heightened worldwide awareness. Malware developers utilise various techniques that are constantly developing to accomplish their objectives. However, obfuscation is most often the first method malware developers use to avoid detection and evade system defences, enabling malware to execute uninterrupted. Therefore, examining this technique is essential for gaining a deeper comprehension of how the malware operates, and the challenges system security engineers confront in developing more efficient detection and countermeasure tactics, and the continued difficulty of keeping ahead of malware authors.

1.1 Objectives

The objective of this thesis is to provide a foundation for understanding what security measures and, more specifically, what obfuscation techniques malware employs to avoid detection and analysis. Furthermore, to acquire good context information, this thesis researches the current state of advanced malware, its implications for existing security solutions, and their ability to detect and defend against rapidly moving malware threat actors and evolving malware threats.

Based on the set objective, this thesis's primary research question was determined to be:

- *What kind of obfuscation techniques does malware employ to avoid system security measures?*

These secondary research questions were created to answer the main research question:

- *How has malware evolved?*
- *How does malware accomplish its goals?*
- *How have detection and defending techniques evolved?*

The research employed a qualitative research methodology with an integrative literature review to accomplish the set objective. A literature review encompassed a systematic and critical evaluation of technical documents published by security researchers, books and e-books, reports published by various organisations and other relevant sources about the research topic. An integrative literature review aimed to identify, evaluate, and synthesize existing knowledge and research findings to establish a foundation for the present study. In addition, the objective was to collect the most recent knowledge on the researched topic.

The thesis focuses on recent advancements in malware development, particularly obfuscation techniques and their relationship with modern technology. Due to the vast amount of information available on the topic, the scope has been restricted to the most relevant and up-to-date information. The thesis assumes that the reader has some technical expertise. However, the goal is to present the research findings clearly so they can be comprehended without prior knowledge. In addition, the scope of this study does not include a comprehensive analysis of the technical complexities of malware. Instead, the study will focus on analysing current and past obfuscation techniques and exploring potential mitigation strategies. The thesis begins by discussing the history and theory of malware, summarizing how it has become the favourite cybercrime weapon and part of a billion-dollar business model for cybercriminals. Next, different obfuscation techniques were researched in detail. Moreover, the conclusion discusses emerging trends, provides valuable malware protection guidelines and gives a good understanding of future threats.

1.2 Ethics

This thesis adheres to the ethical guidelines set by JAMK University of Applied Sciences (*JAMK University of Applied Sciences*, 2018). The research and the resulting content in this thesis were obtained from publicly available sources and did not infringe on any copyrights or reveal confidential information. Because malware technologies are continually developing, material in this thesis for some of the researched techniques may be outdated or otherwise obsolete but are represented to give historical background and to demonstrate the type of evolution that has occurred. This research's content, tools, and examples are presented solely for educational and research purposes. Abuse and unlawful behaviour involving this information is forbidden and may result in penalties, fines, and legal action. All information and sources are cited following JAMK's reporting guidelines (*JAMK*, 2022).

2 Malware

2.1 History

1970s to 1990s

Viruses emerged as computer use increased due to the popularity of the IBM PC and MS-DOS. Compared to today's malware, computer viruses are tiny in size. They often performed activities such as destroying data or altering the PC's BIOS to prevent the computer from starting. Viruses typically propagate further by copying themselves to every floppy disk pushed into the machine (Bettany & Halsey, 2017). A computer virus is a well-known phenomenon that often elicits negative emotions. A computer virus, like the flu, can swiftly transmit from one host to another, creating minor to catastrophic consequences.

The development of malicious or intrusive software is not recent, as demonstrated by the creation of a proof-of-concept virus in 1971 to study viral transmission over ARPANET, the precursor to the current Internet, before its widespread availability. The objective of the test was not to cause harm but to ascertain whether the virus could spread between machines via ARPANET, and it was successful. (Saengphaibul, 2022)

Initially, viruses were relatively harmless and not designed to cause harm or be a catalyst for a nuclear attack. Many viruses and maliciously acting software started as scientific experiments in classrooms or computer labs, intended as simple pranks for their peers. However, by the end of the ’80s, viruses became a worldwide phenom and threat to computer users. (Mercante, 2018)

The Morris Worm, discovered at the end of 1988, is one of the oldest and first viruses to gain significant mainstream media attention. A graduate student at Cornell University (Ithaca, New York) launched Morris Worm from the Massachusetts Institute of Technology computers. Although the virus only intended to assess the size of the Internet for its developer, an error in its programming transformed it from a harmless worm into an infectious denial-of-service tool that required a long time to remove from the thousands of machines it infected. (Bettany & Halsey, 2017)

It did not take long for malicious actors to exploit the concept of malicious software. The world’s first ransomware, the AIDS Trojan, also known as the PC Cyborg Virus, was discovered in 1989. The late Dr Joseph Popp developed the relatively simple virus and spread it via floppy disks sent worldwide by post to AIDS researchers. Unfortunately, when the recipient installed the benign-looking software intended to help AIDS researchers, the installer contained invisible malware that infected the user’s system. Initially, the malware sat dormant until the system booted up for the 90th time, at which point it locked the system and demanded victims pay $189 to a P.O. Box in Panama for access to their system. According to Dr Joseph, the ransomware was to collect funds for AIDS research. However, other reports state that he made it out of frustration with the World Health Organization after being rejected for a job. (Cohen, 2021)

In the 1990s, computer viruses became more advanced and harmful, causing significant damage to the systems they infected. As a new twist, certain viruses were designed to be activated at specific times or in particular circumstances. In 1992, the Michelangelo virus was discovered, which spread via floppy disks. This virus was set to be activated on March 6th, Michelangelo’s birthday, resulting in significant media coverage across TV and print platforms. Consequently, authorities recommended that computer owners either turn off their computers or change their computer’s date to prevent the virus from causing damage. (Saengphaibul, 2022)

As the 90s came to a close, the frequency and severity of virus attacks continued to rise. The main goal of these early viruses was to cause harm to the victim's computer system by either destroying files, preventing the system from booting up or displaying entertaining texts or graphics on the victim's screen. The rise of attacks attributes to the growing number of PC users and the increasing connectivity of the world. (Mercante, 2018)

2000s-2010s

With the arrival of the new century (also known as Y2K), fast-expanding and widely available Internet access connected individuals all over the world, resulting in a dramatic increase in the number of attacks as the number of possible victims increased. Also, the malware and the threat actors behind the attacks shifted from simply damaging victims' systems to profiting from them (Mercante, 2018). Introducing digital currency (also known as cryptocurrency) helped facilitate this shift. One of the pioneers of this technology was Bitcoin, founded in 2008 by an unknown individual or group under the alias Satoshi Nakamoto. Bitcoin is a digital currency that does not have intermediaries such as banks or governments. Instead, transactions are verified via cryptography by network nodes and stored in a public ledger known as a blockchain (Frankenfield, 2022). Cryptocurrency has many legitimate applications, such as facilitating quick and safe cross-border transactions without intermediaries like banks. However, the features that make cryptocurrencies appealing for lawful applications, like decentralisation, anonymity, and lack of regulation, also make them very appealing for criminal activity. Initially, cryptocurrency transactions were largely untraceable. However, in recent times, law enforcement officers have traced some of the transactions back to the threat actors (Ostroff & Vigna, 2020).

In 2010, a state-sponsored Chinese hacking group APT10 launched the Aurora malware attack campaign, widely regarded as one of history's most major cyber espionage efforts. Its objective was to steal intellectual property from large American corporations such as Google, Adobe, and Dow Chemical. The attackers utilised spear-phishing to trick employees into clicking on malicious links or files. Once they had access, they used various techniques, including proprietary malware and backdoors, to maintain persistence and access sensitive information. The success of the attack campaign serves as a stark reminder of the importance of implementing strong cyber security measures and the ongoing need to remain vigilant in the face of constantly evolving cyber threats. (Ali, 2022)

Also, in 2010, the infamous Stuxnet computer worm was discovered, meant to disable a critical component of Iran's nuclear programme. The creators of the Stuxnet malware remain a topic of contention and speculation. However, it is widely accepted that Stuxnet was a joint effort involving US and Israeli intelligence organisations. Stuxnet was a sophisticated malware that used many zero-day vulnerabilities in the Windows operating system. Once infected, it searched for a connection to the software controlling the target device, such as a nuclear centrifuge. Upon finding a target, it transmitted commands to the equipment, leading it to malfunction and finally fail. International Atomic Energy Agency (IAEA) inspectors, who were permitted to access one of the facilities targeted by the attack, noticed an unusually high number of damaged centrifuges, subsequently confirming that the attack had succeeded in its goal, possibly setting back their nuclear program by years. (Fruhlinger, 2022b)

In 2013, a group of hackers known as Carbanak stole almost $1 billion from banks worldwide using custom malware. The attackers began with spear phishing email campaigns, luring people to open the malicious email and infecting PCs with malware. The custom malware provided a backdoor for the attack group to gain a footing on the bank's intranet and travel laterally through the network to discover other PCs to extract funds. (*Kaspersky*, 2015)

In the same year, a hacker infiltrated Target Corporation's (Target), a general merchandise retailer, security and payment systems, resulting in one of the most significant security breaches. The attack was made possible through a phishing email campaign using Citadel, a variation of the Zeus banking trojan virus. The attacker stole 40 million credit card records and 70 million customer records. Target paid $18.5 million to settle claims, with an overall cost estimate of more than $200 million, including losses incurred due to fewer households purchasing at Target following the data breach. Interestingly the attackers breached the target systems through a third-party portal, which allowed them to jump into the target's network. Proper network segmentation and tight access control could have prevented the target's breach. (Jones, 2021)

Ransomware began to appear more frequently in 2013, as seen by Cryptolocker's disastrous ransomware campaign. Because of its powerful encryption technique, Cryptolocker was one of the first ransomware attacks to gain widespread attention. It employed military-grade 2048-bit RSA

encryption, making it nearly impossible to decode victims' files without paying the ransom. (Buckbee, 2015)

In 2014, one of the largest banks in the United States, JP Morgan Chase, suffered a massive data breach. The attackers accessed the bank's systems by installing malware on its servers, stealing personal information from over 76 million homes and 7 million small enterprises. JP Morgan Chase suffered a massive setback due to the data breach, which harmed the bank's brand and resulted in considerable financial losses. The stolen data included names, addresses, phone numbers, email addresses, and other sensitive information, making it a goldmine for cybercriminals. The incident emphasises the significance of adequate access control mechanisms and multifactor authentication. The bank could have avoided the data breach if it had adopted effective access control mechanisms and multifactor authentication. (Kurane, 2014)

In the same year, the Emotet Trojan was first identified by security researchers. The Emotet Trojan is a highly complex malware that infects and compromises systems via various tactics. Emotet's most effective strategy is to send convincing phishing emails to unsuspecting victims, which can deceive users into downloading and installing malware onto their systems. Emotet can steal critical information such as login credentials, banking information, and personal data after it has gained access to a computer. It can also exploit the infected computer to send spam emails, propagate to other network-connected devices, and distribute further infections like banking Trojans. Despite cyber security professionals' efforts to defeat the Emotet Trojan, the malware remains a severe threat to individuals and enterprises. (*Malwarebytes*, 2023)

In 2015, a cyber-attack was directed towards the Ukrainian power grid, resulting in a power outage that impacted up to 230,000 individuals living in the Ivano-Frankivsk region, home to approximately 1.4 million people. The attackers thought to be a Russian organization known as Sandworm, used a combination of malware, spear-phishing emails, and remote access tools to obtain access to three power distribution companies' control systems. They could then remotely disable circuit breakers at 30 substations, causing a blackout that lasted several hours. This attack was the first known instance of a successful cyber-attack on a power grid, highlighting the vulnerability of critical infrastructure to cyber threats. The incident led to increased international cooperation on cyber security issues and prompted a renewed focus on cyber security and collaboration between

governments and industry organizations worldwide to develop best practices and share threat intelligence. The Ukrainian government established a dedicated cyber security centre to improve the country's cyber defences, and the attack remains a watershed moment in the history of cyber security. (Krigman, 2022)

In 2016, a devastating ransomware attack called Petya emerged for the first time. It spread quickly through infected email attachments, and once it had infiltrated a system, it encrypted the victim's files and demanded payment in exchange for the decryption key. However, unlike other ransomware attacks, Petya had a twist: it encrypted the victim's files and overwrote the master boot record (MBR) of the infected system, rendering it unusable. Even if victims paid the ransom, there was no guarantee that they would be able to recover their data or regain access to their systems. As a result, the Petya attack caused widespread panic and chaos, with many businesses and organizations scrambling to protect themselves from the threat. The Petya attack was particularly insidious because it was able to spread so quickly and easily. In addition, it exploited a vulnerability in the Windows operating system, which meant it could infect entire networks by using stolen credentials to move laterally from machine to machine. The attack was initially targeted at Ukrainian businesses and government organizations, but it quickly spread to other countries, affecting companies and individuals worldwide. The Petya attack was a wake-up call for the cyber security industry, highlighting the need for better network hygiene and more robust backup and recovery processes. While subsequent attacks have been even more devastating than Petya, it remains a potent reminder of the power and danger of ransomware. (*Malwarebytes*, 2022a)

In 2017, WannaCry (also known as WannaCrypt or WannaCrypt0r), a ransomware worm, spread rapidly across many computer networks. After infecting a computer, WannaCry encrypted the user's files and demanded a ransom in exchange for the decryption key. The first ransom demanded was $300 in bitcoin, with the threat of increasing the amount if the ransom was not paid within a set date. WannaCry exploited an EternalBlue vulnerability in Microsoft Windows, allowing it to propagate without user input or consent. Interestingly, the WannaCry infection had a built-in kill switch, which cyber security researchers discovered during the outbreak. The researchers found that if a particular URL were registered on the Internet, WannaCry would shut off. The researchers triggered this kill switch shortly after it was discovered, which aided in containing WannaCry infections. The EternalBlue exploit is also notable in the WannaCry attack because it

was allegedly found by the United States National Security Agency (NSA), who built code to exploit the weakness instead of disclosing it. The Shadow Brokers, a hacker outfit, later stole this vulnerability. Following the incident, Microsoft criticized the US government for failing to disclose its knowledge of the issue sooner. (Fruhlinger, 2022)

Also, in 2017, the NotPetya cyber-attack, disguised as a ransomware epidemic, swept through worldwide businesses, inflicting severe harm and costing billions of dollars. The malware originated in Ukraine and exploited a vulnerability in Microsoft's Server Message Block protocol (SMB) to infect multiple countries and critical systems of major corporations and government agencies. Unlike traditional ransomware, NotPetya overwrote the master boot record (MBR) of infected systems, rendering them unusable and forcing affected organizations to rebuild their IT infrastructure from scratch. The attack was believed to be a deliberate act of cyber sabotage and highlights the growing threat of cyber-attacks to global businesses and critical infrastructure. The NotPetya attack underscores the need for organizations to take proactive steps to improve their cyber security posture. Implementing robust backup and recovery procedures, regularly patching software vulnerabilities, and investing in employee training and awareness programs can help mitigate the risk of cyber-attacks. In the wake of the NotPetya attack, many organizations have taken steps to improve their cyber security practices and better prepare for future threats. However, as cybercriminals evolve their tactics and capabilities, the threat of another devastating attack remains real. Therefore, organisations must remain vigilant and adaptable in this ever-evolving threat landscape. (Capano, 2021)

The COVID-19 pandemic had an expected impact also on the cyber security threat landscape. The pandemic forced many people to work from home in a new hybrid office model, impacting how and where people worked. In addition, companies were required to establish connectivity between employees and the company for remote working due to the rapid development of remote work and transfer some critical functions to cloud services for better accessibility. However, this also increased the playground for threat actors and the need to employ and develop more robust defences and educate personnel on cyber security dangers and how to protect devices and network connections in remote environments.

Recent History

According to an annual report released in 2022 by Malware Bytes (an American Internet security-centred company and end-point protection detection software developer), in 2020, there was a significant 24 per cent decrease in malware detections on Windows business machines and also in cybercrime activity due to the Covid-19 pandemic hitting the global economy hard. However, as coronavirus restrictions were gradually lifted globally in 2021, malware returned with unprecedented force, with 77 per cent more malicious software observed than in 2020. (*Malwarebytes Threat Review*, 2022)

In 2020, the SolarWinds cyber-attack, also known as the SUNBURST attack, was the first supply chain type of attack. The hackers were believed to be of Russian origin and gained access to SolarWinds' network by compromising its software build system. The hackers inserted malicious code into a software update distributed to SolarWinds' customers, including numerous government agencies and major corporations. The code remained hidden and allowed the hackers to access the networks of affected organisations, steal sensitive information, and monitor communications. The attack was revealed when cyber security firm FireEye found the malicious code on its network, which led to a broader investigation and eventually told the extent of the SolarWinds attack. The SolarWinds cyber-attack had significant impacts on the affected organisations. The attack resulted in data theft, network monitoring, reputational damage, regulatory compliance concerns, and financial costs. The breach damaged these organisations' trust in their stakeholders and raised concerns about compliance with regulatory requirements, particularly in heavily regulated industries such as finance and healthcare. The attack was costly for the affected organisations, who had to devote significant resources to investigating and remediating the breach. The attack highlighted the need for increased vigilance and robust cyber security measures to protect against future attacks. (Paganini, 2021)

Remote work also meant that mobile device usage raised, and mobile devices increased the attack surface for cybercriminals. Verizon 2022 Mobile Security Index (MSI) report (a report gathered from 632 security professionals around the world) revealed that the new world of hybrid work that has resulted from the transition to remote work had presented security teams with an uphill struggle as the number of devices and remote employees grows. According to the report, 79 per cent of companies have identified that remote working adversely affected their cyber security and

increased the burden on security teams. Also, an alarming 45 per cent of the companies have encountered mobile device breaches, which is twice what was observed in the 2021 survey. The report also indicates a lack of security training (44 per cent of companies do not give adequate security training regularly, and 51 per cent do not provide security-related training when employees, for example, are switching to remote working). Also, lack of remote working guidance is seen as one major problem where only 65 per cent of the companies have guidelines for remote work. (*Verizon*, 2022)

In 2021, the two most essential themes were adaptability and persistence, according to a report released by CrowdStrike (an American Internet security-centred company). Despite the uncertain future, businesses adjusted to the new post-Covid environment with innovative technical solutions. However, while these measures were to enhance and strengthen the situation in the future, they also increased risks and created new vulnerabilities for cybercriminals to exploit. Even though law enforcement provided new methods to thwart illegal activities, cybercriminals continuously responded to changes in the target landscape and successfully operated ransomware. CrowdStrike saw an 82 per cent rise in ransomware-related attacks in 2021 over the previous year. The CrowdStrike report emphasises that adversaries are not resting but rather intensifying attacks which will be more devastating and inflict more comprehensive destruction, affecting employees, small businesses, and large corporations. (*CrowdStrike*, 2022)

Given the reported numbers, it would be easy to conclude that covid-19 was the reason for the significant malware drop in 2020. However, an annual report released in 2021 by ENISA (European Union Agency for Cybersecurity) found that the decline in detections in 2020 can partly contribute to the transition to remote work, which reduced visibility into malware typically found on corporate infrastructure as employees used personal internet connections and devices. (European Union Agency for Cybersecurity, 2022)

In 2022, a conflict between Russia and Ukraine prompted cybercriminal activities by Russian and Ukrainian threat actor organisations. The Russian government pushed cybercriminals to disseminate disinformation and destabilise Ukraine’s president, but their efforts were ineffective in swaying world opinion in their favour. Meanwhile, Ukrainian threat actor organisations have begun disclosing crucial information about how ransomware groups work, perhaps resulting in a permanent

break between Russian and Ukrainian rivals. At the same time, China-based threat actor organisations have been assaulting hardware security solutions in the cyber security and infrastructure industries. (*Sophos*, 2022)

2.2 Theory

The term “Malware” is a combination of “malicious” and “software” (*Kaspersky*, 2021). Malware refers to any programme (software) intended to harm or exploit computer systems. Malware is invasive software purposefully developed to damage a system, steal data, or obtain illegal access. Unlike inadvertent software bugs, malware is intended to undermine a computer system’s Confidentiality, Integrity, and Availability or more commonly referred to as CIA Triad (*Malwarebytes*, 2022b). Information security depends on these three factors to protect data and information from illegal access, alteration, or destruction. The CIA Triad presents confidentiality, integrity, and availability as a framework for developing, implementing, and maintaining the security of computer systems (*Fortinet*, 2022).

Malware has become more common today than viruses, and malware developers are continually devising new malware types and variants. Today, malware includes trojans, spyware, worms, viruses, adware and rootkits. Following is a list of the most common types of malware, each with a brief description:

Adware is a form of malware that displays unwanted advertisements to users, typically as pop-ups or banners. It is frequently distributed as part of free software downloads, but it can also be obtained through a cyberattack or vulnerability. Adware aims to generate revenue for its developers by displaying advertisements to consumers. However, it can cause significant harm to an infected computer by degrading its performance, reducing its availability, and breaking the user's privacy by collecting information about their browsing activities. (*What Is Adware?*, 2022)

Fileless malware is a type of malware that infects a system without using files in the traditional sense. Instead, it utilises legitimate applications or tools already installed on the system. Malware that operates entirely in memory does not require the installation of any files on the victim's computer to infect and harm it. Fileless malware exploits vulnerabilities in software programs, operating systems, or even hardware to gain access to a victim's system. Once inside, the malware

employs various techniques to avoid detection and remain in memory, including injecting itself into other legitimate processes, utilising rootkits, and employing code obfuscation techniques. (Arntz, 2021)

Ransomware is a type of malware that cybercriminals use to encrypt data, typically business critical and valuable information, and demand payment in exchange for a decryption key. Although it has existed since the 1980s, it has become a significant threat since the early 2010s, with attackers using it for financial gain or espionage. With ransomware, the goal of the attackers is to make money. Since victims are often willing to pay a ransom to regain access to their data, it is a viable business strategy for cybercriminals. Defending against ransomware requires a multi-layered approach, including keeping systems and software up-to-date with the latest security patches, performing regular backups, training employees to identify phishing scams and other social engineering techniques, and having a well-defined incident response plan (*What Is Ransomware?*, n.d.). In addition, because attackers may exfiltrate data during a ransomware attack, they may employ a double extortion scheme by threatening to expose the data or sell it to other criminals. Some attackers have evolved the double extortion strategy into a triple extortion scheme by demanding payment from additional individuals who could be harmed if the company's data is leaked (Paskoski, 2022).

Rootkits are malware that can run close to or within the kernel of an operating system, allowing hackers to seize control of the system. They can infect any device with an operating system, including Internet of Things (IoT)-connected appliances such as refrigerators and thermostats. Rootkits can be used to conceal keyloggers, allowing cybercriminals to steal sensitive information such as credit card numbers and online banking credentials. In addition, they can launch DDoS attacks, send spam emails, and disable or remove security software. Some rootkits are used for legitimate purposes, but most are used maliciously to transmit malware that modifies the operating system and grants remote users administrative privileges. Rootkits are challenging to detect and eliminate, making them a serious threat to computer security. (*How to Detect & Prevent Rootkits*, n.d.)

Spyware is malicious software that secretly collects and transmits information about users' devices and online activities. There may also be legal spyware software that monitors data for advertising or commercial purposes. However, malicious spyware is expressly designed to profit from

stolen data. Whether legitimate or fraudulent, spyware surveillance activities expose sensitive data to abuse and data breaches. In addition, spyware can degrade the speed of devices and networks, slowing down everyday activities. Therefore, it is essential to be aware of the potential threats of spyware and take precautions to protect privacy and security. (*What Is Spyware?*, 2022)

Trojans are malicious software that disguises as legitimate programmes or files and can take control of a computer to execute malicious operations such as stealing data, damaging the system, or opening backdoors for other viruses. The term "Trojan" derives from the Greek legend of the Trojan horse, a deceitful tactic used to conquer Troy. In the digital realm, Trojans are hostile digital parasites capable of reading passwords, recording keystrokes, and spreading other viruses. Social engineering techniques, such as phishing emails or malicious downloads, are used to spread them. Unlike computer viruses and worms, Trojans cannot replicate and must be installed or executed by users. (*What Is a Trojan Horse and What Damage Can It Do?*, 2023)

Viruses are malicious software that spreads from computer to computer, intending to cause damage by interfering with systems, causing operational difficulties, and causing data loss or leakage. Computer viruses attach themselves to executable host files and spread through networks, files, file-sharing applications, and infected email attachments. Common computer virus symptoms include a slow system, uninvited pop-up windows, self-executing programs, logged-out accounts, and system crashes. (*What Are Computer Viruses?*, n.d.)

Worms are a type of malware that can spread across networks without the intervention of a host programme or a human. They are malicious trojan horses that can replicate themselves and spread from one computer to another. Worms infect their hosts through deception and cleverness, and they can cause severe damage to compromised computers by consuming bandwidth, overloading systems, deleting or modifying files, and installing additional viruses. In addition, vulnerabilities in software, email attachments, network connections, and portable devices can facilitate their propagation. They are especially harmful due to their tendency for self-replication and network propagation. (*What Is a Computer Worm?*, n.d.)

Threat actors are driven by a variety of motivations to create malware. They may seek to earn money, disrupt operations, make a political statement, or display their abilities. Even though

malware does not typically cause physical damage to devices, it can still steal, encrypt, or destroy victims' data and take control, lock, or modify the compromised device's functions, causing significant harm. Cybercriminals frequently employ malware to monitor computer activities, collect data, and prepare for future attacks. (*Malwarebytes*, 2022b)

Developing malware and its components requires specialized expertise. As detection and countermeasures improve, attackers constantly evolve their malicious code to adapt to their victims' changing environment. Furthermore, threat actors engage in activities such as trading, selling, stealing, improving, and reusing malware, making it difficult for researchers and law enforcement to track them. The emergence of numerous malicious programs, new malware variants, and strains makes it a continuous and unequal struggle to defend against attacks and discourage threat actors. (European Union Agency for Cybersecurity, 2022)

Virus creation was initially viewed as a way for computer specialists to demonstrate their technical and programming skills. As time passed, however, it became a tool used for more malicious purposes, such as the theft of sensitive information such as credit card numbers, passwords, and bank account information, or for personal revenge. At the onset of the evolution of viruses, no techniques to evade detection by code analyzers or specialists attempting to identify malicious code existed. Despite this, computer programmers were thrilled to discover new ways to create viruses and test the limits of their programming skills. As viruses became more complex, programmers were able to create techniques to make them more challenging to detect and analyze, resulting in the development of four generations of stealth techniques (also known as obfuscation techniques). These techniques were Encryption, Oligomorphism, Polymorphism, and Metamorphism (see Figure 1). The gradual evolution of these obfuscation techniques enabled virus authors to evade detection and continue to create increasingly sophisticated viruses. (Bashari Rad et al., 2012)

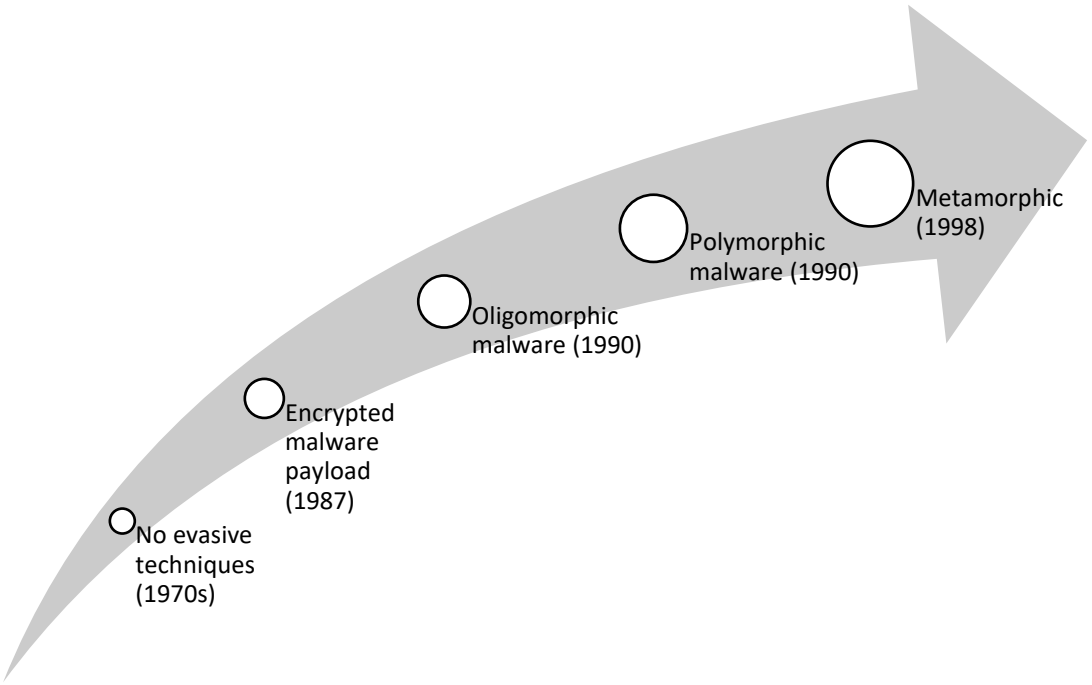


Figure 1. Evolution of evasion techniques employed in malware

In cyber security, a never-ending conflict exists between the individuals responsible for creating malware and those striving to defend against it. Unfortunately, those who make malware often have the upper hand in this constant struggle. Despite the concerted efforts of the cyber security industry to combat this issue, the frequency and complexity of malware attacks persistently increase, indicating that malware creators have been consistently beating security defenders in their pursuit of malicious objectives.

2.2.1 Malware and technique classification

Malware and its functionalities differ depending on the phase or situation in which the malware is used in a cyber-attack. Therefore, several ways for classifying malware have been developed. Malware, for example, may be classified using taxonomy, a technique for detecting and categorising malware based on characteristics such as origin, kind, structure, behaviour, and target platform (Tripathy et al., 2018). Malware can also be classified using static and dynamic analysis, where static analysis examines the malware features without executing the malicious code, and dynamic analysis examines the behaviour of active malware in a simulated environment. Another way to classify malware is to use machine learning (Mahajan et al., 2019).

Malware can also be classified as traditional malware and advanced malware. In the past, traditional malware was commonly found everywhere and did not focus on specific targets. It typically exploited known vulnerabilities and was not designed to remain hidden for extended periods. Additionally, these attacks were often short-lived and did not significantly threaten the affected system. However, with the evolution of technology, malware has become much more sophisticated. Advanced malware tends to be precisely targeted, often created with a specific target in mind and is very persistent. Advanced malware can also take advantage of previously unknown vulnerabilities, known as “zero-day” (0-day) vulnerabilities and very complex methods to evade detection. Therefore, advanced malware is a more dangerous and challenging threat to deal with than its counterpart. (Singh & Singh, 2018)

Additionally, obfuscation techniques can be classified according to their potency, resilience, and cost, where potency measures the difficulty for humans to comprehend the obfuscated code, resilience measures the difficulty of reverting the code to its original form, and cost indicates the impact on execution time and application size (Collberg et al., 1997). Finally, obfuscation techniques can also be classified based on their sophistication, detection difficulty, and impact, where sophistication measures the technical level of obfuscation, detection difficulty measures the difficulty in detecting obfuscated code, and impact measures the effort required by vendors to enhance their systems to counter the threat (Marpaung et al., 2012).

Classifying and organising the various techniques and methods used when conducting research or experiments is essential. Classifying (or categorising) facilitates analysis and comparison of the various techniques and identifies patterns and trends. This study acknowledges the significance of classification and, where applicable, lists the various types of classifications for each technique. By providing classification, it enables readers to have a better understanding of the various techniques used and their interrelationships, making it easier to draw meaningful conclusions from the study's findings and, in addition, facilitating the potential replication of the techniques by other researchers or their application in a variety of contexts. Table 1 summarises obfuscation technique classifications as potency, resilience, cost, sophistication, detection difficulty and impact on a scale from Low to High.

Table 1. Summary of obfuscation technique classifications

Technique	Potency	Resilience	Cost	Sophisti-cation	Detection difficulty	Impact
Code transposition	High	Low	Low	High	Low	Low
Compression	High	Medium	Low	High	High	Medium
Dead-code	Low	Medium	Low	Medium	Medium	Low
Encoding	Medium	Medium	Low	Low	Medium	Low
Encryption	High	High	Medium	High	High	Medium
Indirect method call	High	High	Low	High	High	Medium
Instruction substitution	Varies	Varies	Varies	Varies	Varies	Varies
Non-alphanumeric code	High	Medium	High	High	Low	Low
Polymorphism	High	High	Medium	High	High	High
Randomization	Low	Medium	Low	Low	Medium	Low
Register randomisation	Low	Low	Low	Low	Low	Low
Return-Oriented Program-ming	High	Medium	Low	High	High	High
Self-modifying code	High	Medium	Low	High	High	High
String splitting	Medium	Low	Medium	Low	Medium	Low
Whitespace decoding	High	High	High	High	High	High
Whitespace randomisa-tion	Low	Low	Low	Low	Low	Low

2.2.2 Detecting and protecting from malware

Today, signature-based methods are one of the most common ways antivirus software detects malware. This method compares the source code of potentially malicious files against a database of known malicious signatures. Nonetheless, cybercriminals frequently employ code obfuscation techniques to change the malicious signature, conceal the true nature of their malware and avoid

detection by these antivirus tools, which means relying solely on signature-based antivirus software to detect and prevent malware infections may not be sufficient. (Mumtaz et al., 2021)

In order to mitigate the potential damage caused by malware attacks, the detection and prevention of malware is paramount. Anti-virus scanners, anti-malware scanners, and endpoint protection systems are among the solutions developed for this purpose. These tools employ diverse techniques such as static analysis, behavioural analysis, sandboxing, and heuristics to detect and prevent malware from executing on a system. Combined with routine software updates and user training, these tools offer valuable assistance in safeguarding computer systems against malware threats.

Anti-virus scanners identify and classify potentially harmful files, programmes, and processes using static, dynamic, and heuristic methods. Static identification is based on previously known identifiers that match malware material or parts of malware content, whereas dynamic identification is based on recognising previously known malicious behaviours and activity. Both of these methods rely on previously collected information on existing malware, and therefore a heuristic detection was developed to attempt to find previously undiscovered and known malware using patterns and small code blocks of existing known malware or techniques. (Abraham, 2017)

Endpoint protection systems were designed to work in combination with traditional anti-virus software to strengthen cyber security protocols in business environments. These systems are designed to protect against cyber threats on various digital devices, including but not limited to personal computers, laptops, mobile phones, and tablets. The endpoint protection solutions include a software application that actively monitors and blocks any security incidents on the device. The program then reports the incidents to a centralised management console. Managing and monitoring numerous devices from a centralised location enables IT administrators to streamline security operations and enhance their oversight and regulation. In addition, implementing these solutions reduces the probability of extensive security breaches, as cyber security risks can be promptly detected and addressed efficiently.

Despite the existence of sophisticated anti-virus and endpoint security systems, malware developers maintain an unfair advantage. By putting their newly developed malware through continuous

testing against existing anti-virus, anti-malware and other protection tools, the malware developers create distinctive versions for the malware not detectable by current detection technologies. Thus allowing new malware variants to breach computer systems before detection mechanisms are updated, providing malicious actors with a window of opportunity to execute effective campaigns.

Malware developers have a further advantage of using newly discovered zero-day (0-day) vulnerabilities, which they either purchase on the dark web or discover themselves. By exploiting previously unknown security vulnerabilities in software and operating systems, they can create malware that exploits these vulnerabilities. These vulnerabilities are called zero-day because the software developer has not discovered or patched them yet, giving attackers a window of opportunity. Using zero-day vulnerabilities, malware developers can create malicious software that can infect systems, steal data, or gain unauthorised access (also known as privilege escalation) without being detected by security software or antivirus programmes. Employing zero-day vulnerabilities enables them to outpace security measures, avoid detection, and cause significant harm to individuals and organisations. Consequently, detecting and patching zero-day vulnerabilities is essential for preventing cyber-attacks and protecting sensitive data.

Due to the unequal competition between those who create malicious software and those who protect against it, the field of information security faces a significant challenge. Attackers continuously modify their techniques, approaches, and protocols, whereas security personnel must adapt to prevent intrusions. Even though security measures may assist in reducing risks, they are not entirely adequate, and security personnel must maintain constant vigilance to counteract new threats. To maintain a level of agility comparable to that of their adversaries, security professionals must be vigilant and prepared for any unforeseen security issues that may arise.

Researchers have developed new Artificial Intelligence-based methods to combat the seemingly impossible task of identifying previously unidentified new malware. Artificial Intelligence (AI) is a fascinating technological phenomenon that almost all businesses would like to take advantage of because it allows for cost savings, such as replacing humans with artificial intelligence in jobs that could previously only be performed by humans. Artificial Intelligence can classify, detect, and prevent computer viruses and malware using Machine Learning techniques. (Faruk et al., 2021)

Deep learning is an innovative technique recently used to overcome the limitations of existing malware identification and categorization techniques. Because it can analyse large data sets and recognise patterns and characteristics that may not be readily apparent to human analysts or conventional machine learning algorithms, deep learning is a powerful technique for identifying malware. Deep learning models use artificial neural networks to acquire knowledge from vast datasets, allowing them to identify sophisticated deviations in code or behaviour that may indicate the presence of malicious software. In addition, unlike conventional techniques that rely on rule-based methodologies or signature matching, deep learning can adapt to innovative and evolving forms of malware. This feature makes it an effective tool for identifying previously unknown threats and maintaining an advantage over malicious actors who continually develop new evasion techniques. As a result, deep learning techniques offer a more advanced and effective technique for identifying and mitigating malware. (Aslan & Yilmaz, 2021)

2.2.3 Malware analysis

Static and dynamic are the two main categories of malware analysis. By scanning, the virus signature, programme structure, and executable form are extracted during static analysis. The executable binary is then converted into machine code using reverse engineering to analyse and identify known malicious code usages, patterns, and structures. On the other hand, in dynamic analysis, malware is executed in a controlled virtualized environment while its interactions are monitored. Observations during the analysis process include creating, modifying or deleting protected files and processes and initiating data transfers. Based on these findings, analysts can determine the behaviour and intentions of the malware, identify its network communication patterns, and ultimately develop countermeasures to prevent its spread and neutralize its effects.

Identifying and mitigating the threat posed by malware requires static and dynamic analysis techniques. Static analysis helps identify patterns and structures of known malicious code, whereas dynamic analysis can detect new and unknown malware. The combination of these techniques provides a comprehensive understanding of the behaviour and intent of the malware, which is required to develop effective mitigation measures. Malware analysts, therefore, frequently employ a combination of these techniques to identify and analyse malware, thereby protecting systems and data from harm caused by malicious software. (Singh & Singh, 2018)

Malware that employs sophisticated obfuscation techniques can be extremely challenging to analyse through static analysis alone. In such cases, dynamic analysis can be a valuable method to uncover the ultimate intentions of the malware by studying its behaviour. While the static analysis may provide valuable details about the structure and pattern of malicious activity, the dynamic analysis offers crucial information on the actual conduct of malware within a controlled environment. By utilising dynamic analysis, experts can learn how the malware interacts with the environment, the system calls it makes, the files it modifies, and the network traffic it generates. This information can be critical for developing effective strategies to counteract the threat posed by malware. (Singh & Singh, 2018)

Malware authors and attackers aim to create obstacles for analysts by using armouring and evasion techniques to make detecting and analyzing malware difficult. Armouring techniques hinder malware analysis, while evasion techniques evade anti-malware tools. Although there is no clear distinction between the two, these techniques are commonly used to identify artefacts that suggest the presence of an analysis environment or tool.

Malware identifies these artefacts using different methods and alters their behaviour, making them challenging to detect and analyze. One such method involves analysing the system to detect virtualization software such as VirtualBox or VMware and adjusting its behaviour accordingly. Additionally, malicious software may look for particular files or registry entries that indicate the existence of a virtualized system. In addition, some malware might detect virtualization by analysing the system's operational efficiency or network communication for anomalies. If the malware detects that it is operating in a virtualized environment, it can either cease execution or demonstrate harmless behaviour to evade detection, making it difficult for analysts to identify and neutralise the threat. (Mohanta & Saldanha, 2020)

2.2.4 Script-based and native malware

Historically, viruses were distributed in the form of native machine code, meaning that they were directly compiled into executable files that could be run on specific computer systems. Typically, these viruses were spread via infected floppy discs, email attachments or online downloads. However, in recent years, JavaScript and other script-based malware have become more widespread due to the growing popularity of the Internet and web-based applications.

JavaScript, a script-based programming language, has rapidly become the most popular language on the web. JavaScript is the backbone of practically every website on the Internet, and its popularity has steadily increased over the last decade. It is now supported by all contemporary web browsers and is implemented on 95 per cent of all websites (2020). In addition, it is consistently listed in the top 10 most popular programming languages, and hundreds of libraries and frameworks have been created around it. JavaScript is now compatible with all processor architectures, allowing the creation of a new platform for malware. In addition, JavaScript is robust enough to operate on mobile devices and desktop programmes that use the same JavaScript code base as their web-based counterparts. (Herrera, 2020)

JavaScript is a high-level programming language widely used to create interactive web pages and is currently present on nearly every website on the internet. Unfortunately, malicious actors employ JavaScript to launch malware attacks against unsuspecting users by inserting malicious JavaScript code into web pages using various methods. These methods may include exploiting security flaws in the website, injecting malicious code into third-party plugins, or even employing social engineering techniques to convince users to execute the code. Once the malicious code is executed, it may steal sensitive information, redirect users to malicious websites, or download additional malware onto the victim's computer. To make matters worse, by utilizing JavaScript, malicious actors can also exploit zero-day vulnerabilities, which are undiscovered software security flaws, or target users with obsolete web browsers to ensure the successful execution of their malware.

JavaScript-based attacks have been identified as the greatest threats to Internet security. Attackers can conduct various attacks by exploiting multiple vulnerabilities in various Web applications, such as cross-site scripting (XSS), cross-site request forgery (CSRF), and drive-by downloads. Most Internet users rely on anti-virus software to guard against malicious JavaScript code. Unfortunately, obfuscation techniques frequently compromise the effectiveness of anti-virus software. Malicious JavaScript code increasingly employs obfuscation techniques to evade anti-virus software detection and conceal its malicious intent. (Xu et al., 2012)

During the initial attack phase, malicious actors typically utilise multiple scripting languages best suited for the attack phase and its goals. One such scripting language is Power Shell, initially designed to automate tasks for Windows system administrators. Due to its efficiency, accessibility,

simplicity of obfuscation, and ease of access to system resources, have become an attractive tool for malware developers. Power shell scripts are frequently used in the initial phase of an attack to download additional malware onto a system and are occasionally combined with macro scripts within Word documents. (Liu et al., 2018)

In addition to Powershell, several other scripting languages are known to be used for malicious goals. For example, python is a popular choice for malware development due to its user-friendliness, robust libraries, and cross-platform compatibility. Other popular choices are Ruby, Perl, and Bash scripting languages capable of developing malware, as they offer a variety of features for malicious activities such as keylogging, data theft, and network scanning. However, it is essential to note that these languages are commonly used for legitimate purposes and are not inherently malicious.

WebAssembly is another language that is extensively supported by browsers. WebAssembly was initially announced in 2015, and by 2017, it was already supported by all major browsers. As of September 2022, WebAssembly was present in 96% of all browser installs. WebAssembly is an efficient compilation target for computation-intensive libraries written in languages like C and C++. In addition to its suitable applications, WebAssembly provides malware developers with additional means of evading detection (Romano et al., 2022). While converting JavaScript to WebAssembly is practically impossible, which is why WebAssembly has never been intended as a substitute for JavaScript but rather as a supplement (Haas et al., 2017). Recent research into this topic shows that replacing carefully selected pieces of JavaScript functionality with WebAssembly counterparts is feasible, which helps avoid malware detectors without compromising code correctness. Malware authors could, for example, use WebAssembly to obfuscate string literals, control-flow statements, and array initialisations (Romano et al., 2022).

Script-based malware is written in a high-level language (such as JavaScript, VBScript, or PowerShell script), distributed in script form, and can operate on multiple platforms without code changes. Depending on the scripting language and the environment, script-based malware might be interpreted on the fly or compiled into machine code before its execution (also known as on-demand compilation). This compilation step is usually optional and transparent and does not change how the malicious code operates but is used to speed up the execution close to its native

counterpart. Two popular browsers, Microsoft Edge and Google Chrome, use the ahead-of-time (AOTC) and just-in-time (JITC) compilation methods to speed up the web page javascript execution speed. (Park et al., 2015)

Even though script-based malware has been seen very commonly in attack campaigns, native malware is still frequently applied after the initial attack phase. Native malware refers to malware written in a low-level programming language (such as C and C++) that is transformed (using a compiler tool) into machine code before distribution. Native malware can only execute on the platform for which it was compiled, so a different version of malware must be compiled to attack different architectures, such as PC, Mac, or mobile platforms. Native malware excels in execution speed, has direct access to system resources, can be concealed using sophisticated obfuscation techniques and does not need additional software such as a web browser to be executed.

It is essential to note that most of the researched techniques apply to native and script-based malware. All the researched native code (also known as machine code) examples, assembler source code, and disassembly of executable code are shown using the Intel assembly instruction set. There are currently two mainstream processor architectures: Intel/AMD, which uses the Intel x86 instruction set and is a Complex Instruction Set Computer (CISC) processor, and Apple, which uses ARM architecture and is a Reduced Instruction Set Computer (RISC) processor. Both architectures are susceptible to malicious attacks, although different processor architectures can make creating and analysing malware more difficult.

3 Obfuscation

Obfuscation is a method that transforms or modifies plain, easy-to-read and understandable code, script or text into a new version that is purposefully difficult to understand and reverse-engineer for researchers and automated analysis. Legitimate software developers typically use obfuscation to protect their intellectual property, making it harder for others to copy or modify their code, prevent reverse engineering of the software, or infringe on copyright licenses. However, malware developers commonly use obfuscation to make their code more difficult to detect and evade security measures. Using obfuscation techniques, malware authors can disguise their malicious code, making it more difficult for researchers to analyze, detect, and mitigate. (O'Kane et al., 2011)

Reverse engineering is a process of disassembling something to understand how it works to replicate, improve, or learn (Schwartz, 2001). However, reverse-engineering malware code is challenging and time-consuming, especially with hardened or designed to resist analysis. These strategies make it difficult for analysts to comprehend the behaviour and functioning of the code, hence hindering reverse engineering attempts. Such program-hardening approaches fall under the broad category of obfuscation (Dang et al., 2014).

Software developers use obfuscation techniques as one of the most effective methods against malicious reverse engineering, making it more challenging to interpret and reverse engineer by hackers and pirates. Obfuscation can play a role in security through obscurity (STO) by making it more difficult for attackers to understand the inner workings of a system and more challenging for attackers to find vulnerabilities or weaknesses to exploit. However, relying solely on obfuscation techniques for security can be dangerous, as it can create a false sense of security and is not a complete solution for preventing software piracy. More robust solutions that implement a combination of technical and legal measures, such as digital rights management (DRM) and licence keys, are available. These techniques make it more difficult for a pirate to understand and construct a working copy of the code. With enough time and resources, determined and talented pirates will discover a method to reverse engineer the code. (Lee et al., 2012)

Malware authors use obfuscation techniques to generate new malware variants from the same malicious code and to evade antivirus and antimalware detections. The availability of various evasion techniques to malware attackers increases the challenge of preventing source code piracy and malicious attacks while struggling to decompile the malware application packages for further analysis. In addition, the fear of encountering difficulties in malware reverse engineering motivates researchers to use evasion techniques to secure the code of benign applications. (Elsersy et al., 2022)

Generally, commercial anti-virus and anti-malware solutions are based on signature-based scanning and heuristic analysis. In signature-based scanning, the signature generated from files is compared to an extensive database where all previously found signature identifiers are stored. The effectiveness of using signatures to detect malware is limited to previously known malware, but it is fast and has a low false positive rate compared to heuristic analysis techniques. This limitation

prompted the development of heuristic analysis to discover potentially harmful programme components. Heuristic analysis is a technique for detecting computer viruses, including new virus variations, by executing suspicious applications in a virtual environment and monitoring them for infectious activity. Heuristic analysis can also decompile and analyse the code for frequent malicious instructions. While heuristic analysis is improving its capacity to detect new viruses, its usefulness is restricted due to the number of false positives, and it may overlook new infections with previously unseen code. (Lin & Stamp, 2011)

Multiple different obfuscation techniques exist, and mixing and layering various techniques together is possible. This research investigated numerous obfuscation techniques from as many different categories as feasible. The following sub-chapters cover the following types of obfuscation categories:

- *Behavioural Obfuscation Techniques*
- *Code Modification Techniques*
- *Code Transformation Techniques*
- *Compression Techniques*
- *Control Flow Obfuscation Techniques*
- *Data Obfuscation Techniques*
- *Encoding and Encryption Techniques*
- *Function Obfuscation Techniques*
- *Whitespace Obfuscation Techniques*

Unfortunately, there is not currently a standard vocabulary that could be employed to categorise various obfuscation techniques, although different techniques can be classified, for example, according to the purposes for which each obfuscation method is employed. Security researchers, intrusion responders, and threat intelligence analysts frequently use the ATT&CK framework maintained by MITER Corporation in the cyber security sector to understand the tactics and techniques used by threat actors and create efficient defences against them (*MITRE ATT&CK Framework*, 2022).

The MITRE ATT&CK is a framework that helps to categorize and comprehend the cyber adversaries' tactics and techniques based on real-world observations. It provides a standardised

taxonomy for building efficient cyber security defences and developing specific threat models and methodologies in various industries. The Matrix categorises the methods and techniques of threat actors into tactics, techniques, and mitigation measures, making it easier for organisations to detect and prevent attacks. It guides how organisations can implement countermeasures to prevent or detect attacks against each tactic and technique. The MITRE ATT&CK Matrix continues to be a valuable resource for organisations seeking to improve their security posture and protect themselves from a wide variety of cyber threats. (*The Mitre Corporation*, 2023)

The MITRE ATT&CK framework focuses on high-level information and, as such, does not provide an extensive amount of specifics about various obfuscation techniques. However, obfuscation strategies and techniques have been grouped under three primary tactics: T1140 Deobfuscate/Decode Files or Information, T1027 Obfuscated Files or Information, and T1001 Data Obfuscation. See Appendix 1 on page 103 for a graph illustrating the Mitre ATT&CK matrix, highlighting the obfuscation techniques and the sub-techniques.

In the framework, each technique, tactic, and mitigation is described in detail; for instance, the T1027 Obfuscated Files or Information is located in the Defence Evasion category and contains the following (condensed for easier reading) and is shown in Table 2.

Table 2. Mitre ATT&CK information about Obfuscated Files or Information

Mitre ATT&CK information about Obfuscated Files or Information	
Main category	**Defence Evasion [TA0005]**
Sub category	**Obfuscated Files or Information [T1027]**
Sub techniques	T1027.001 Binary Padding T1027.002 Software Packing T1027.003 Steganography T1027.004 Compile After Delivery T1027.005 Indicator Removal from Tools T1027.006 HTML Smuggling

Mitre ATT&CK information about Obfuscated Files or Information			
	T1027.007	Dynamic API resolution	
	T1027.008	Stripped Payloads	
	T1027.009	Embedded Payload	
Tactic	Defence Evasion		
Platforms	Linux, Windows, macOS		
Defense Bypassed	Application Control, Host Forensic Analysis, Host Intrusion Prevention Systems, Log Analysis, Signature-based Detection		
CAPEC ID	CAPEC-267		
Contributors	Christiaan Beek, @ChristiaanBeek; Red Canary		
Version	1.3		
Created	31 May 2017		
Last Modified	30 September 2022		
Mitigations (4)	M1049	Anti-virus/Antimalware	
	M1040	Behavior Prevention on End-point	
Detection (5)	DS0017	Command	Command Execution
	DS0022	File	File Creation & Metadata
	DS0011	Module	Module Load
	DS0009	Process	OS API Execution & Process Creation

Mitre ATT&CK information about Obfuscated Files or Information			
Summary	"Adversaries utilise numerous approaches, such as encrypting, encoding, or obfuscating the contents of executables or files, to make them hard to identify and analyse. To evade detection, payloads may also be compressed, archived, or encrypted and may require user action or passwords to open. Adversaries may disguise plain-text strings by using compressed or archived scripts and encoding sections of files. They may also break payloads into seemingly harmless files that, when reassembled, disclose malicious activity. To avoid detection, adversaries may use platform-specific semantics to disguise instructions run from payloads or command and scripting interpreters." (*The Mitre Corporation*, 2023)		
Procedure Examples	ID	Name	Description
	S1028	Action RAT	"Action RAT's commands, strings, and domains can be Base64 encoded within the payload"
	S0045	ADVSTORESHELL	"Most of the strings in ADVSTORESHELL are encrypted with an XOR-based algorithm; some strings are also encrypted with 3DES and reversed. API function names are also reversed, presumably to avoid detection in memory."
	…		
Mitigations	**ID**	**Mitigation**	**Description**

Mitre ATT&CK information about Obfuscated Files or Information			
	M1049	Antivirus/Anti-malware	Anti-virus can be used to automatically detect and quarantine suspicious files. Consider utilizing the Antimalware Scan Interface (AMSI) on Windows 10 to analyze commands after being processed/interpreted.
	M1040	Behavior Prevention on Endpoint	"On Windows 10, enable Attack Surface Reduction (ASR) rules to prevent execution of potentially obfuscated payloads."
Detection	**ID**	**Data Source / Component**	**Detects**
	DS0017	Command Execution	"Monitor executed commands and arguments containing indicators of obfuscation and known suspicious syntax such as uninterpreted escape characters like ^ and '. Deobfuscation tools can be used to detect these indicators in files/payloads."
	DS0022	File Creation / File Metadata	"Detection of file obfuscation is difficult unless artefacts are left behind by the obfuscation process that are uniquely detectable with a signature. If detection of the obfuscation itself is not possible, it may be possible to detect the malicious activity that caused the obfuscated file (for example, the method that was used to write, read, or modify the file on the file system)."

Mitre ATT&CK information about Obfuscated Files or Information			
			"Monitor for contextual data about a file, which may include information such as name, the content (ex: signature, headers, or data/media), user/ower, permissions, etc." "File-based signatures may be capable of detecting code obfuscation depending on the methods used."
	DS0011	Module Load	"Monitoring module loads, especially those not explicitly included in import tables, may highlight obfuscated code functionality. Dynamic malware analysis may also expose signs of code obfuscation."
	DS0009	API Execution / Process Creation	"Monitor and analyze calls to functions such as GetProcAddress() associated with malicious code obfuscation." "Monitor for newly executed processes that may attempt to make an executable or file difficult to discover or analyze by encrypting, encoding, or otherwise obfuscating its contents on the system or in transit."

Although the framework essentially only provides high-level information about various obfuscation techniques, it can still be used to discover additional information, such as procedure examples which list a high number of known malware instances known to be using obfuscation in some way. The framework also describes tactics, techniques and possible mitigations associated with obfuscation techniques. For example, T1027 Obfuscated Files or Information, at the time of research, listed over 300 procedure examples, nine sub-techniques, two main mitigation strategies and four different detection strategies.

Overall, the Mitre ATT&CK framework is a valuable tool for cyber security researchers and analysts. It provides a comprehensive and structured approach to comprehending and classifying the various tactics, techniques, and procedures employed by attackers, making it easier to identify potential threats and develop effective defence strategies. In addition, the framework is regularly updated with new information and insights based on the most recent threat intelligence, ensuring that it remains relevant and valuable in a threat landscape that is constantly evolving.

The following chapters of the research delve into obfuscation techniques utilized in malware programs to conceal their intended purpose and functionality. First, the study conducts a comprehensive analysis of the different obfuscation techniques, which includes evaluating their purpose, mechanism, robustness, and potential weaknesses. Next, the study presents a detailed explanation of each method and practical examples to help readers understand how each technique operates. Additionally, the study focuses on detecting and mitigating obfuscation techniques employed by malware programs. Finally, the study identifies possible countermeasures to these techniques and examines future developments in malware obfuscation. Overall, the study provides a deeper understanding of malware programs' obfuscation techniques and provides readers with the knowledge to identify and mitigate such threats in the future.

3.1 Code transposition

Code transposition (also known as code reordering, code restructuring, and subroutine reordering) is an obfuscation technique initially used to improve the performance and efficiency of software programs. Table 3 displays classification information for the technique. Code-reordering involves rearranging the order of instructions in a program's code to make it more challenging to analyse and understand while retaining functionality. Code reordering has legitimate uses in software development, such as optimising and protecting software from other attackers and hiding sensitive information. Unfortunately, malware authors frequently employ this method to avoid detection and analysis by antivirus, antimalware, and security experts. (Cimitile et al., 2017)

Table 3. Code transposition classification

Technique	**Code transposition**
Aliases	Code reordering, code restructuring, subroutine reordering
Categories	Code Transformation Techniques Control Flow Obfuscation Techniques
Potency, resilience and cost	
Potency	High
Resilience	Low
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	High
Detection difficulty	Low
Impact	Low

In legitimate software development, code-reordering is a valuable optimization technique that enables processors to execute software more efficiently. This optimization occurs during the compilation phase when the compiler transforms the program code into machine code. The compiler employs various optimization techniques, including code reordering, to ensure the most efficient

execution of the program. By predicting the most common execution sequence of the program, the compiler rearranges the code accordingly, allowing the processor to operate the most efficiently (Brais, 2015)

Malware authors use manual and automatic code reordering techniques since they make analysis harder and generates numerous unique signatures for the same malware. However, at the same time reordering code technique is also used to generate numerous malware variants, each with a unique signature, making it more difficult for antivirus and antimalware software to detect and block the malicious code. (Elsersy et al., 2022)

Multiple code-reordering techniques exist, including; Instruction reordering: Shuffling around instructions or groups of instructions without changing the program functionality, and Subroutines reordering: Changing the order of the execution of subroutines. Known malware employs at least two distinct ways. The first approach randomly shuffles the instructions and then inserts jumps between now-shuffled instructions to maintain the correct program flow (See Figure 2).

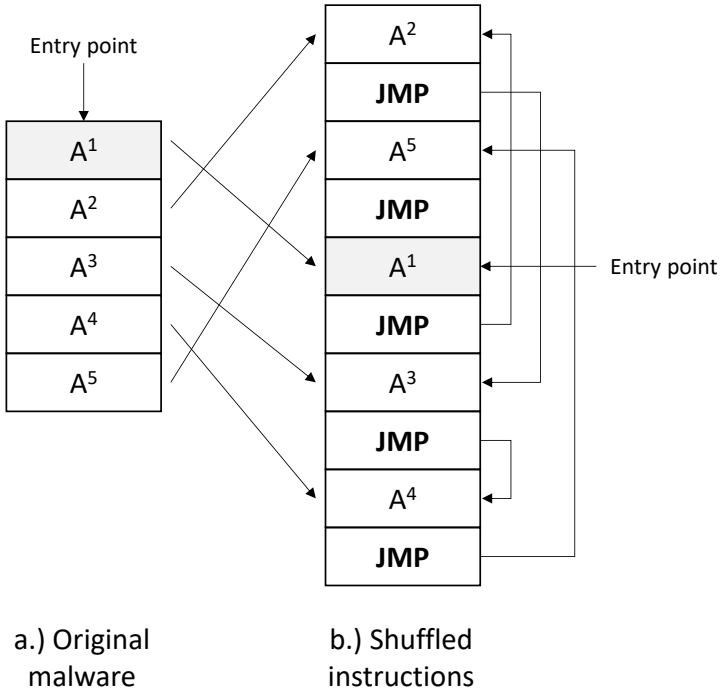


Figure 2. Code reordering by shuffling instructions

The second technique shuffles independent instructions, code blocks or subroutines (meaning they do not influence one another). Programs often call several subroutines in a specific order. As a result, generating an enormous (n!) number of distinct variations of the same malware is possible by shifting these subroutines around. Figure 3 shows an example of shuffling ten different subroutines. In this example, the execution starts from the first (1) subroutine. (You & Yim, 2010)

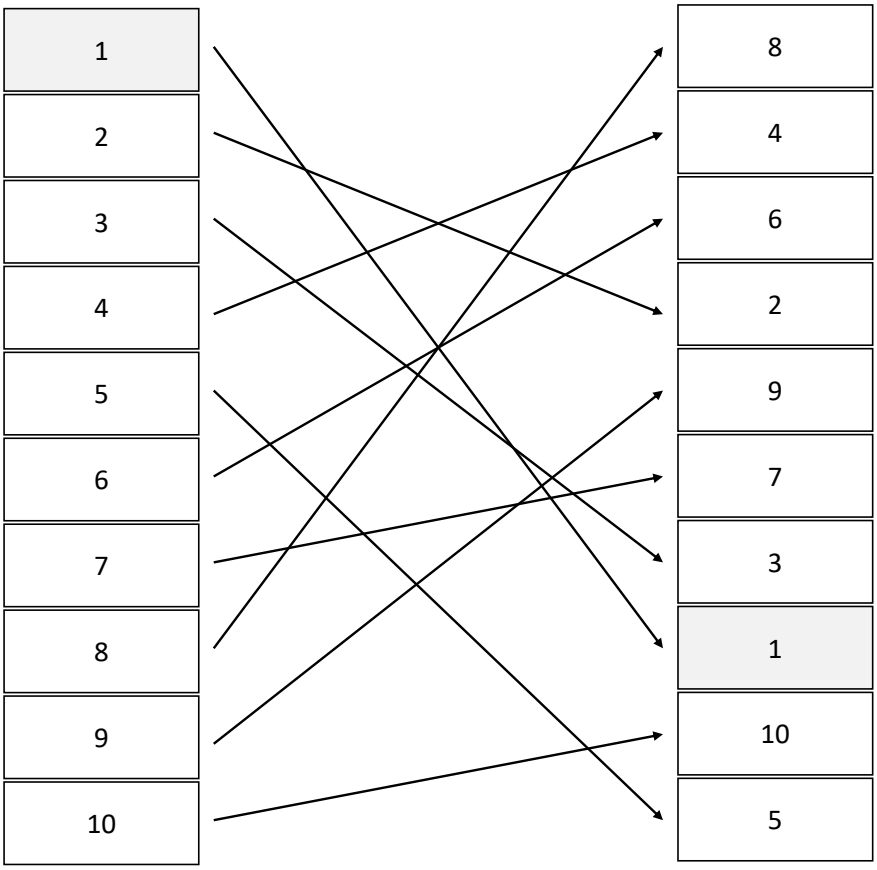


Figure 3. Example of subroutines (1-10) reordered to create distinct variation.

It is also possible to reorder independent code blocks, as seen in Figure 4. These two similar techniques can also be combined. For example, one known malware employing subroutine shuffling is Ghost Backdoor, which has ten separate subroutines and produces a maximum of 10! = 3628800 distinct versions.

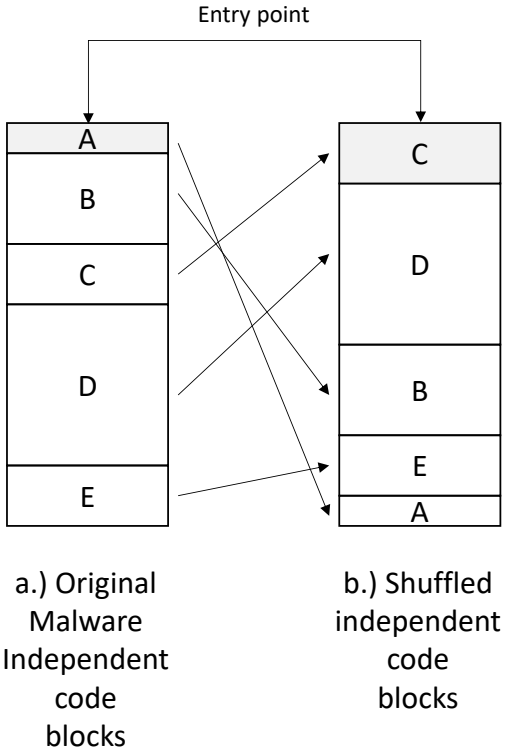


Figure 4. Example of shuffled code blocks.

Static analysis of native code can uncover malicious intent by examining the use of various API calls. Because code reordering does not modify the contents of the Import Directory Table, which contains entries for every Dynamic Link Library loaded by the executable, it may be analysed to determine if the program is malicious. For example, specific imports such as cryptography, file reading and writing, file searching, and so on might indicate a malicious program. Likewise, static analysis of script-based code can uncover malicious intent by examining the use of various key-words. Code reordering does not affect dynamic analysis, so it is possible to mitigate with stand-ard dynamic execution prevention techniques used by anti-virus and anti-malware software. How-ever, these methods may result in difficulty in grasping and comprehending code. (You & Yim, 2010)

3.2 Compression

Compression (also known as packing) is an obfuscation method initially developed to minimise memory and bandwidth for storing and transmitting data and is now frequently used in the digital business world. Table 4 displays classification information for the technique. However, due to the nature of the compression process, even a minor change in the source file's contents substantially affects the compressed file and its signature. Malware authors exploit this characteristic by employing various algorithms to compress malicious code, thereby reducing its size and making it easier and quicker to distribute, but making it more difficult for security researchers and antivirus or antimalware software to identify and analyse the malware. (O'Kane et al., 2011)

Table 4. Compression technique classification

Technique	**Compression**
Aliases	Packing
Categories	Compression Techniques Data Obfuscation Techniques
Potency, resilience and cost	
Potency	High
Resilience	Medium
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	High
Detection difficulty	High
Impact	Medium

Malware developers frequently use compression techniques to evade antivirus software detection and security measures. Compression obfuscates the malware's true nature and enables it to infiltrate computers undetected. Given this, compression is an essential technique for these malicious actors, and a comprehensive understanding of compression algorithms is crucial for preventing and identifying malware infections. Furthermore, without this information, it would be challenging

to create effective countermeasures to protect computer systems from these types of threats. Thus, recognising the importance of compression in creating malware and implementing countermeasures against it are essential steps in protecting computer systems from malicious attacks.

Compressed (packed) malware was one of the first methods for evading signature-based anti-virus scanners. Figure 5 shows compressed malware that typically consists of the unpacker (also known as a stub) programme and the packed payload (the malware in compressed form). When the compressed malware is executed, the unpacker obtains the harmful payload, decompresses it, and executes it in memory. (*Packed Malware*, 2020)

In this technique, the malware is first compressed and then prepended with a decompressor program (Unpacker in Figure 5). The malware author may use standard compression, modified compression, a proprietary compression method, or a mix of different methods. Some compression algorithms additionally allow using a password to encrypt the payload while compressing the payload.

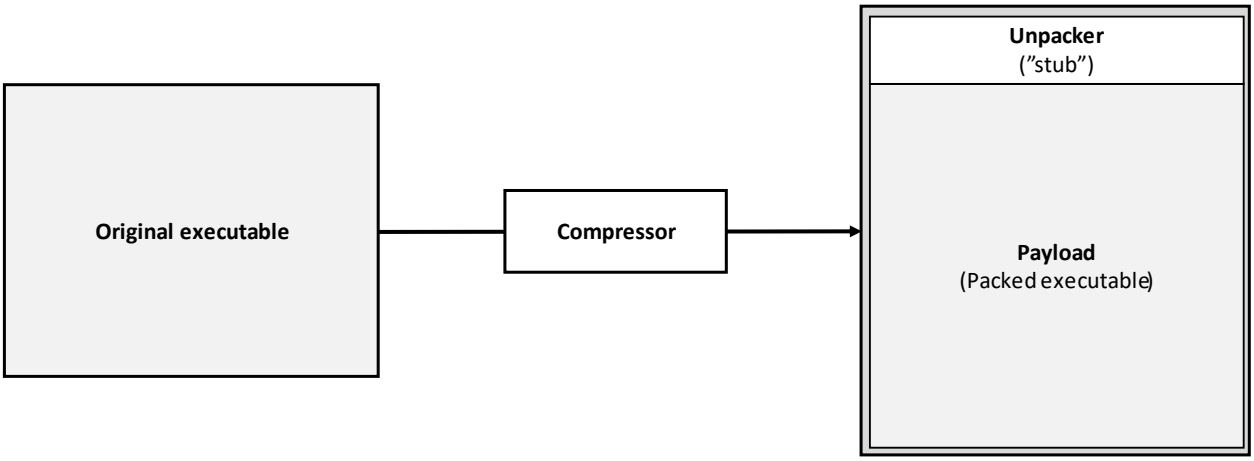


Figure 5. Compressed executable

The fundamental shortcoming of this method is that the unpacker remains constant from generation to generation. As a result, anti-virus scanners may detect this type of malware based on the unpacker's coding pattern. In most cases, a compressed payload makes automatic static analysis of the payload impossible unless the compression is one of the known compression schemes and no encryption is used. Using dynamic analysis, it is possible to analyse the unpacked contents of the packed payload while it is uncompressed in memory. Also, malware researchers can

potentially manually unpack the malicious payload using the known unpacker and, optionally, by utilising the password key stored with the unpacker or brute-forcing compression with different standard algorithms. Unpacking the payload this way avoids the need to execute the malicious payload code, which could be dangerous.

While it is possible to use compression in script-based malware, it is not that commonly used and would be inefficient because of the slower execution speed of the script-based implementation. More commonly, an uncompressed (but obfuscated) script downloads the payload, usually a malware native code executable that is compressed and obfuscated.

3.3 Dead-code

Dead code (also known as garbage code) is an obfuscation technique used to enhance software security by making it challenging for attackers to comprehend how a proprietary software system works. Table 5 displays classification information for the technique. Dead code is code that does not affect the execution of the program and is inserted between regular code, making it harder for reverse engineers to analyze the source code and understand the program flow. Dead code has many legitimate uses, such as protecting software from piracy. However, malware writers often use dead code to make it difficult for reverse engineers to understand malicious code, increasing the overhead of reverse analysis. (Wang et al., 2017)

Table 5. Dead-code technique classification

Technique	**Dead-code**
Aliases	Garbage code, Junk code, Zombie code
Categories	Behavioural Obfuscation Techniques Code Transformation Techniques Control Flow Obfuscation Techniques
Potency, resilience and cost	
Potency	Low
Resilience	Medium
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	Medium
Detection difficulty	Medium
Impact	Low

In native malware, authors use the dead-code technique to modify program signatures while keeping the original functionality intact. The NOP (No Operation) instruction, which does not perform any operations and does not affect how the malicious programme behaves, is a typical

example of a dead-code instruction in native code. A malicious attacker can alter the programme's signature without impacting its functioning by injecting NOP instructions between lines of existing code. This method is frequently employed since the modified programme can seem to differ from its original signature, which makes antivirus software less likely to pick it up. Although dead-code instructions may help conceal malicious code, they are easily recognised by skilled analysts, making them less reliable for evading security measures. (You & Yim, 2010)

In script-based malware, using dead code is an effective and easy way to create variations of the same malware without causing any disruption to the malware's original execution algorithm. In most cases, this is accomplished by adding code blocks that are never executed because of false circumstances or by introducing short anonymous function blocks known as lambda instruction blocks. These lambda instruction blocks may be provided as input to other functions or preserved in variables. An adversary might add code to malicious software using these approaches, although the added code may not directly influence the software's functionality. These methods might disguise the true purpose of malware and make it more challenging for security researchers to identify and analyse. (Xu et al., 2012)

An example of javascript code in Figure 6 shows how the original javascript code (a) could be modified by inserting dead code blocks (functions and lambda instruction blocks) between javascript instructions (b), making the resulting code more challenging to understand and analyse while also providing a different and possibly previously undetected signature for the malware.

```
function show_alert(text) {
    alert(text);
}
var hello = ‘This could be malicious';
show_alert(hello);
```

(a)

```
// Dead code
function a() { b(); }
function b() { a(); }
function show_alert(text) {
    alert(text);
}
// Garbage code
function c(f) { f(); }
() => {var p = 'Checked by Antivirus'};
var hello = ‘This could be malicious';
// Garbage code
c(() => {var p = 'Checked by Antivirus'});
show_alert(hello);
```

(b)

Figure 6. Dead code injection in JavaScript

Similarly, native malware might be obfuscated by inserting dead code instructions (such as NOP instruction that does not perform anything) between machine code instructions. Figure 7 shows the original assembly code, and Figure 8 shows the code after it has been modified.

```
public start
start proc near
push    rbp                                     ; 55
mov     rbp, rsp                                ; 48 89 E5
sub     rsp, 20h                                ; 48 83 EC 20
call    sub_140001284                           ; E8 77 02 00 00
lea     rcx, aHelloWorld  ; "Hello world! \r\n" ; 48 8D 0D EC 1F 00 00
call    sub_140001038                           ; E8 1F 00 00 00
xor     rax, rax                                ; 48 31 C0
call    ExitProcess                             ; E8 0F 00 00 00
start endp
```

Figure 7. Original program without obfuscation

```
public start
start proc near
push    rbp                                     ; 55
nop     <-                                      ; 90
mov     rbp, rsp                                ; 48 89 E5
nop     <-                                      ; 90
sub     rsp, 20h                                ; 48 83 EC 20
nop     <-                                      ; 90
call    sub_140001284                           ; E8 74 02 00 00
nop     <-                                      ; 90
lea     rcx, aHelloWorld  ; "Hello world! \r\n" ; 48 8D 0D E8 1F 00 00
nop     <-                                      ; 90
call    sub_140001038                           ; E8 1A 00 00 00
nop     <-                                      ; 90
xor     rax, rax                                ; 48 31 C0
call    ExitProcess                             ; E8 09 00 00 00
start endp
```

Figure 8. Obfuscated code with dead code (code lines marked by arrows)

Simple script instructions or native instructions that perform nothing are easy to identify, whereas identifying code blocks irrelevant to the malware is considerably more difficult. Furthermore, even though the malicious functionality of the programme remains the same across different versions, the resulting signatures are different; therefore, static signature-based detection is insufficient. Instead, dynamic analysis is required for accurate malware identification by executing the executable within an emulation environment so that malicious components of the executable code can be detected and by determining which malware is involved and how to protect the system from its harmful effects.

3.4 Encoding

Encoding is an important method for preventing data corruption during transport by changing it to a new format. Table 6 displays classification information for the technique. Different systems may employ different character sets, leading to misinterpretation and data corruption during processing. Encoding is often used in computer communications to avoid this issue by ensuring data transfer in a format that the receiving device can accurately process. Encoding allows for safe and exact data transmission between devices, assuring its integrity and dependability. Encoding has numerous practical applications, but the malware also uses it to disguise data or malicious code. Malware often uses encoding to obfuscate data or code since it is simple and quick but difficult to detect or decode automatically. (Kılıç et al., 2019)

Table 6. Encoding technique classification

Technique	**Encoding**
Categories	Data Obfuscation Techniques Encoding and Encryption Techniques
Potency, resilience and cost	
Potency	Medium
Resilience	Medium
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	Low
Detection difficulty	Medium
Impact	Low

Encoding has far-reaching roots. Julius Ceasar, for example, used primitive encoding around 100 BC to hide crucial communications from those who were not supposed to comprehend the message's content, such as when transmitting a message from one person to another. This encoding (or simple encryption) was known as "Ceasar cypher", which functioned by rotating each letter of

the alphabet forward three positions, such that A became D, B became E, and X became A. (Sikorski & Honig, 2012)

Today, encoding acts similarly to how it did in the past, but with more complexity and combining different encoding strategies is possible. Attackers choose encoding techniques that best match their goals. For example, they may use simple cyphers or basic encoding functions, which are straightforward to implement and provide enough protection. Alternatively, they may use complex cryptographic cyphers or specialised encryption to make identifying and reverse engineering more difficult. (Sikorski & Honig, 2012)

Encoding plays a crucial role in Internet communication by ensuring data is transmitted and interpreted correctly across various devices, platforms, and languages. Encoding enables the smooth transfer of text, images, videos, and other digital content by transforming data into a standardised format, like Unicode or ASCII. It enables various systems, regardless of the user's location or preferred language, to comprehend and process data consistently. Additionally, encoding makes it easier for people, companies, and organisations to exchange information globally, fostering effective communication, teamwork, and the free exchange of ideas online.

Characters can be represented in various ways in JavaScript and other script-based languages and resource files. Character escaping is a common technique that involves representing a character in a manner other than simply typing it into the code. This technique was initially intended for non-ASCII characters, but it can also be used to escape conventional characters, allowing the obfuscation of any text or code. JavaScript supports a variety of character escapes, including *Hexadecimal*, *Unicode*, and *Octal*. *Hexadecimal* escapes represent a character using a backslash followed by two hexadecimal digits, whereas *Unicode* escapes use the “u” prefix followed by four hexadecimal digits. *Octal* escapes, on the other hand, represent a character with a backslash followed by up to three octal digits. Overall, character escaping provides programmers with an effective means for representing characters in various contexts and can be used to ensure that various systems and platforms correctly interpret text and code. (Heiderich, 2011)

Base64 encoding is a popular method for transforming binary data into ASCII string format. It is especially beneficial when sending binary data over channels that only support text-based data,

such as email or HTTP. Base64 encoding transforms every three bytes of input data into four bytes of output data consisting of characters from a predefined set of 64 characters. These 64 characters consist of the uppercase and lowercase letters *A* through *Z*, the numerals *0* through *9*, and the symbols "*+*" and "*/*". In addition, a padding character "*=*" is utilised to ensure that the output data length is a multiple of four characters. Commonly, Base64 encoding is used to encode binary data such as images, audio, and video files, as well as executable malware that cannot be transported or stored in binary format. The encoded data may be transmitted or embedded wherever ASCII strings are permitted, including script files and data URLs. Base64 encoding is a trustworthy and effective method for transmitting binary data over text-based channels. (Heiderich, 2011)

Cybercriminals frequently use the XOR operation to obfuscate malware code and evade detection by security software. XOR is a straightforward and effective algorithm for converting binary data, such as strings and executable files, into an unrecognisable format. In turn, static analysis tools cannot identify the malicious code, allowing the malware to circumvent security measures that rely on detecting specific patterns. Recently, attackers have added encryption techniques to their XOR obfuscation strategies. For example, Locky ransomware employs JavaScript XOR obfuscation to decrypt the downloaded payload, making it more difficult to detect and analyse. A second instance of attackers employing XOR encryption was observed in the payload of the "backdoor.Stegmap" malware concealed within a Microsoft Windows logo image file on a public GitHub server. The payload was encrypted with XOR and then exploited exchange server vulnerabilities. As cybercriminals continue to adapt their methods, security researchers and software developers must maintain vigilance and adapt to the most recent techniques and strategies to detect and prevent malicious attacks effectively. (Heiderich, 2011)

To escape a character in JavaScript, the value in hexadecimal, Unicode or octal must be prefixed by a specific character or characters. Hexadecimal escaping starts with the characters "\x" followed by a two-digit hexadecimal value, Unicode escaping starts with the characters "\u" followed by a four-digit hexadecimal value, and octal escaping starts with the character "\" followed by one to three digit value. Examples of different hexadecimal, Unicode and octal values for regularly used characters can be seen in Table 7.

Table 7. Example of escaping characters using hexadecimal, Unicode and octal escape types

Character	Hexadecimal	Unicode	Octal
a	\x61	\u0061	\141
(	\x28	\u0028	\50
)	\x29	\u0029	\51
(space)	\x20	\u0020	\40
(tab)	\x09	\u0009	\11
(non-breaking-space)	\xa0	\u00a0	\240

It is permitted to mix different escape types within the same JavaScript code, even within a single line, which complicates the analysis of malicious scripts. In this example, a simple JavaScript code calls the alert function with a “This could be malicious” text string. The original code without obfuscation can be seen in Figure 9, and the same script obfuscated using the hexadecimal escaping is seen in Figure 10. As seen from the escaped version, the script needs to use helper functions to complete: First, *unescape* is used to deobfuscate the string, and then the resulting string is executed with the *eval* command making both the non-obfuscated script and the obfuscated script behaviorally the same.

```
alert('This could be malicious');
```

Figure 9. Example code without obfuscation

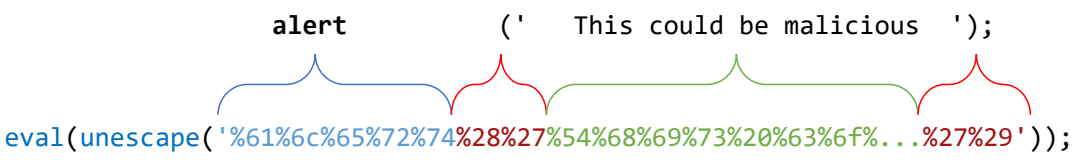


Figure 10. Example code with hexadecimal escapes

In the world of cyber security, identifying various known encodings is a fundamental task. This process involves decoding different types of encodings to detect malicious activities carried out by cybercriminals. Generally, malware developers rarely invent new encoding techniques; instead, they

rely on existing ones to conceal their actions. However, decoding multi-level encodings with automated approaches can be challenging, especially when combining several distinct encoding methods. In such cases, a more sophisticated and customized approach may be required to decode the encodings accurately. Standard signature-based mitigation is an effective way to detect known encodings as it relies on pre-existing signatures to detect malicious activities. However, it may not be as effective against new and unknown encodings, making it necessary to develop new techniques to counteract emerging threats. Overall, identifying known encodings and decoding them accurately is crucial in the fight against cybercrime and in ensuring the safety and security of online systems and networks.

3.5 Encryption

Encryption (also known as cryptography) is an obfuscation technique that leverages the same powerful technique used, among other things, to protect internet traffic and personal information from eavesdroppers or software from piracy and hackers. Table 8 displays classification information for the technique. Cryptography plays a central role in the wide range of daily transactions we carry out with the devices we use. Safe internet communication or phone calls would not be possible without cryptography. This is especially critical for sensitive activities such as online banking and shopping, where consumers must securely send personal and financial data. (Cobb, 2004)

Table 8. Encryption technique classification

Technique	**Encryption**
Aliases	Cryptography
Categories	Data Obfuscation Techniques Encoding and Encryption Techniques
Potency, resilience and cost	
Potency	High
Resilience	High
Cost	Medium
Sophistication, detection difficulty and impact	
Sophistication	High
Detection difficulty	High
Impact	Medium

Encrypted malware was one of the first techniques to avoid signature-based anti-virus scanners. Similarly to the compression technique, any change in the source code will substantially affect the encrypted output and the signature. An encrypted malware generally consists of the decryptor program (often referred to as a stub) and the encrypted payload (the malware in an encrypted form), as seen in Figure 11. When the encrypted malware executes, the decryptor retrieves the

malicious payload, performs decryption and executes the resulting executable code. (You & Yim, 2010)

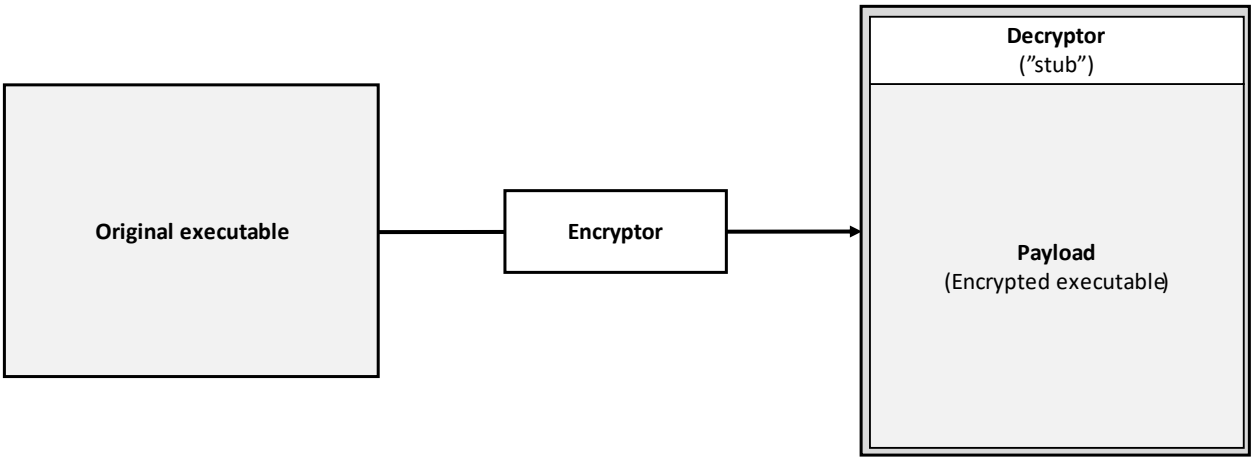


Figure 11. Encrypted malware

The malware author might encrypt the payload with a new key every time to avoid signature-based detections. However, the fundamental shortcoming of this method is that the decryptor remains constant from generation to generation. As a result, anti-virus scanners may detect this type of malware based on the decryptor's coding pattern. To overcome this limitation, malware authors have started using polymorphic techniques that generate new decryptors for each generation of malware. This makes it much harder for antivirus software to identify and block the malware. (You & Yim, 2010)

In most cases, an encrypted payload makes automatic static analysis of the payload impossible. However, malware researchers can manually decrypt the malicious payload using the known decryptor and a found decryption key. The decryption key is typically stored within the decryptor, but it is possible to brute-force decryption using different standard algorithms if it is not found. Decrypting the payload this way avoids the need to execute the malicious code, which is dangerous.

Static analysis can discover the decryptor (the stub) and the decryptor's lengthy binary payload. Static analysis may also examine the import table to see whether standard cryptography APIs are utilised. Dynamic analysis can identify the encrypted payload in memory before or after it is injected into the memory of another process.

3.6 Indirect method call

The indirect method call is an obfuscation technique that exploits a programming technique that enables attackers to execute arbitrary code through a function pointer. Table 9 displays classification information for the technique. A security researcher Matt Kolisar, a DEFCON 16 speaker, unveiled this approach by describing how JavaScript code might use this technique to call any function indirectly bypassing standard detection methods employed by antivirus and antimalware scanners. It includes altering the function pointer to refer to an arbitrary function in memory and then calling it via the function pointer, allowing them to take control of the program and potentially execute arbitrary code on the target system. This approach circumvents several security protections and highlights the need to safeguard and adequately verify function pointers in software applications. (Kolisar, 2008)

Table 9. Indirect method call technique classification

Technique	**Indirect method call**
Categories	Behavioural Obfuscation Techniques Control Flow Obfuscation Techniques Function Obfuscation Techniques
Potency, resilience and cost	
Potency	High
Resilience	High
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	High
Detection difficulty	High
Impact	Medium

In programming languages like C and C++, indirect method calls are often employed to provide dynamic and flexible programme behaviour. This was the first time, however, that this approach was

implemented in a Script-based programming language. After the code has been de-obfuscated manually or automatically, JavaScript is usually expected to have visible function names. Using this strategy in Script-based programming languages, however, entirely conceals function names until they are called, and even then, only the interpreter or browser knows the name of the called function. Calling a method indirectly complicates the detection required to identify these sorts of attacks.

In JavaScript, it is possible to write text to the current document using the *write* method of the current document context. For example, calling "*document.write*('This could be malicious');" would add "This could be malicious" text into the current document. However, malicious code that might not want to reveal that it is using a *document* and its *write* method could employ an indirect call technique.

In the example in Figure 13, the program first searches for the *document* element (there are eight characters in the "document", and the fifth character is m (In programming, the index starts from 0, so the fifth character is found at index 4). Then when the *document* element is found, the program continues by searching for a *write* method whose length is five characters, the third character is "i", and the fifth character is "e". Finally, after the *write* method has been located, the code indirectly calls the *"write"* method with the "This could be malicious" parameter, effectively hiding any indication that it is using the document's *write* method. The full call graph is shown in Figure 12.

Figure 12. Indirect method call graph

```
for (o in this) {
    if (o.length == 8 && o.charAt(4) == 'm') {
        for (m in this[o]) {
            if (m.length == 5 && m.charAt(2) == 'i' && m.charAt(4) == 'e') {
                this[o][m]('This could be malicious')
            }
        }
    }
}
```

Figure 13. Example of obfuscated method calling

This technique is challenging to mitigate because the code does not appear to perform any malicious actions until the final command initiated by the code executes. In order to determine the actual command that the code will execute, dynamic analysis with different techniques is required. One such technique is to use behaviour-based detection methods that analyze the code's behaviour during execution and look for suspicious actions or sequences of actions. Another approach is to use machine learning algorithms to identify patterns in code that are indicative of indirect method calls. Additionally, implementing security measures such as sandboxing and virtualization can help isolate the malware and prevent it from causing harm to the system.

3.7 Instruction substitution

Instruction substitution is an obfuscation technique used in software development to hide the purpose and logic of code from attackers and to protect software from piracy. Table 10 displays classification information for the technique. This approach involves substituting original instructions with semantically comparable ones that produce the same outcomes. The central processing unit (CPU) processor includes various instructions that may be substituted to provide the same output but with a different signature. One of these instructions is the *XOR*, which the *SUB* instruction may replace, and the *MOV* instruction, which a combination of the *PUSH* and *POP* instructions can replace. Two instruction substitution strategies are commonly known. The first approach shuffles instructions and adds unconditional branches or jumps. The second technique creates new copies of the code by reordering separate instructions that do not influence one another. It is challenging to implement these strategies, but once applied, it may make it more challenging to discover and reverse-engineer the original programme code. (You & Yim, 2010)

Table 10. Instruction substitution technique classification

Technique	**Instruction substitution**
Categories	Code Modification Techniques
Potency, resilience and cost	
Potency	Varies by method (potentially very high)
Resilience	Varies by method (potentially very high)
Cost	Varies by method (potentially very high)
Sophistication, detection difficulty and impact	
Sophistication	Varies by method (potentially very high)
Detection difficulty	Varies by method (potentially very high)
Impact	Varies by method (potentially very high)

In this technique, the malware creator exchanges instructions and then creates a new malware based on the newly altered malware code, which is then subsequently deployed. The example shown in Figure 14 uses the instruction substitution technique to modify the original machine

code (a) by replacing the mov instruction with push and pop, the lea instruction with mov and the sub instruction with add (b). These replaced instructions perform the same function but change the resulting code signature.

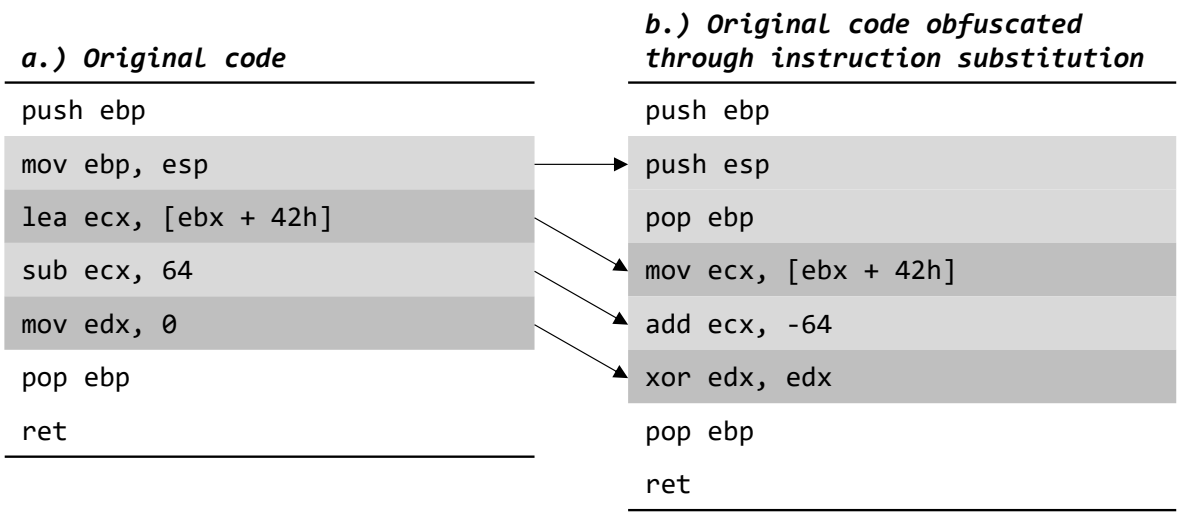


Figure 14. Instruction substitution technique

A more advanced version of instruction substitution was introduced in a research conducted by Stephen Dolan in 2013. The research showed that it is possible to replace every instruction (demonstrated using x86 machine language) with only *MOV* instructions (Dolan, 2013). This approach was eventually called “MOVfuscator”, and tools to automate the process were made available. Since the resulting code is very tough to analyse, this approach offers high potency, resilience, and cost.

To perform MOVfuscation, a special version of the C-language compiler called MOVfuscator is required. It is not currently possible to convert existing compiled (binary) programs into MOVfuscated versions. However, it is possible to compile other languages like C++ into MOVfuscated versions by first compiling them into bytecode, which can then be converted back to C language. The MOVfuscator compiler compiles source C code into Object-code, an intermediate representation which can then be linked generally to an executable.

The following code example is a simple program that only displays the text string ”This could be malicious”. The non-obfuscated compiled version is 15960 bytes (~16 Kbytes), but the obfuscated version is 10,221,284 bytes (~10 Mbytes) which is over 640 times larger. The original non-

obfuscated version can be seen in Figure 15, and the resulting obfuscated program contains only MOV instructions in Figure 16.

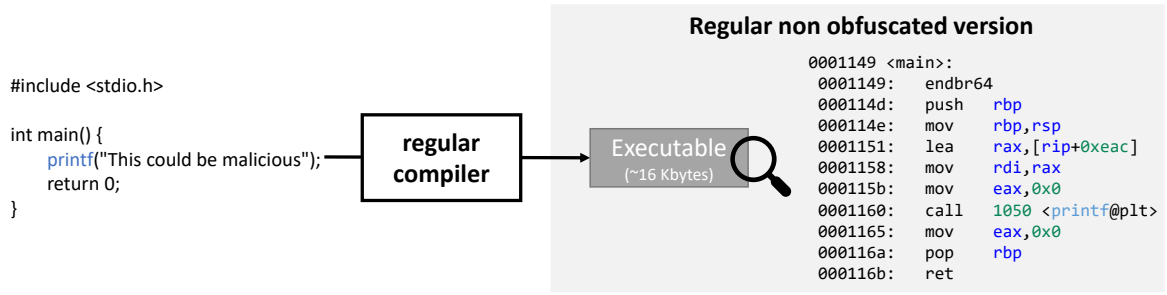


Figure 15. Non obfuscated version

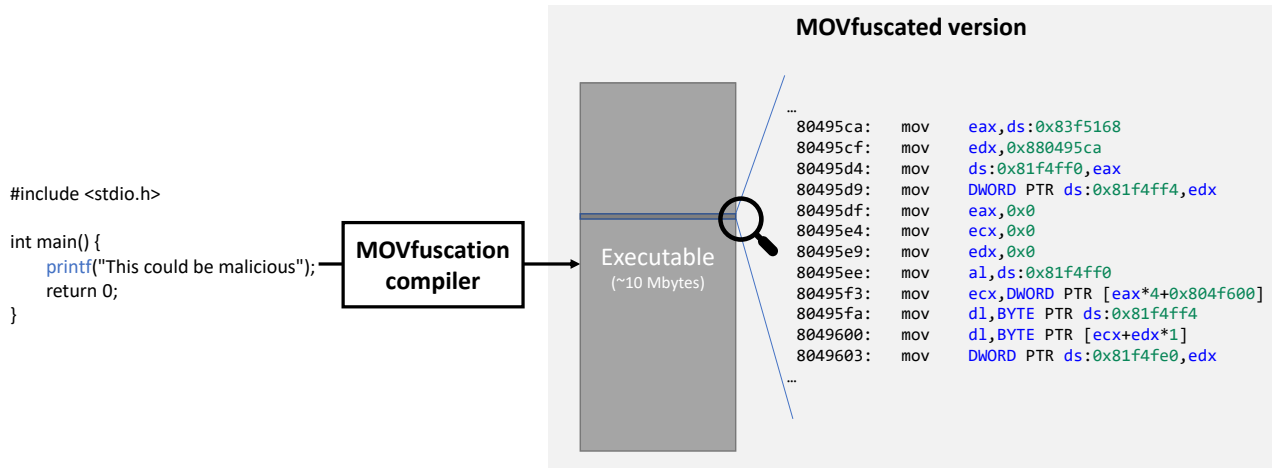


Figure 16. MOVfuscated version

Understanding the program flow using static analysis for the MOVfuscated version is nearly impossible because every line in the program appears to transfer data using mov instructions, and the program flow is not visible. The resulting code is so massive that manual inspection is unrealistic, even for a small example program.

Although researchers and anti-virus system developers seem to confront an impossible task in analysing or deobfuscating *MOVfuscated* malware, it was demonstrated that creating tools that allow decoding the original program's control flow is possible, which may be enough to determine if the software is harmful at a high level. One such attempt was made by Julian Kirsch and Clemens Jonischkeit when they presented the *deMOVfuscator* tool. In their presentation at Recon 2016 (a computer security conference held annually in Montreal, Canada), they describe a method for recovering the original programme's control flow from *movfuscated* binaries without assuming

register allocations or a specific instruction order. This method adheres to the high-level invariants that every *movfuscated* binary must satisfy and is unaffected by proposed hardening techniques such as register renaming and instruction reordering. This is accomplished using a combination of static taint analysis and a satisfiable modulo theory (SMT) solver on the *movfuscated* code. Multiple *movfuscated* binaries have been successfully decrypted with the *demovfuscator*, demonstrating its ability to decrypt real-world binaries. The authors are actively advancing the *demovfuscator* and working towards the next objective of eliminating instruction substitution and producing a more compact and readable output. They intend to share their perspectives on this subject as well.

3.8 Non-alphanumeric code

Non-alphanumeric code is an obfuscation technique used to convert code into a form containing only non-alphanumeric characters. Table 11 displays classification information for the technique. Non-alphanumeric refers to a string of characters that is not made up of letters, numbers, or punctuation marks, which are the most frequent types of characters used in programming languages. The resulting non-alphanumeric code has little resemblance to conventional code, making static analysis very difficult. The origins of non-alphanumeric coding are assumed to be the Obfuscated C and Obfuscated Perl contests, in which competitors strive to write the most obfuscated code possible. These challenges were created to demonstrate how inventive programmers might be in concealing typical source code using generic syntax. (Heiderich, 2011)

Table 11. Non-alphanumeric code technique classification

Technique	**Non-alphanumeric code**
Categories	Code Modification Techniques
Potency, resilience and cost	
Potency	High
Resilience	Medium
Cost	High
Sophistication, detection difficulty and impact	
Sophistication	High
Detection difficulty	Low
Impact	Low

Malware developers often use this method to make it more difficult for antivirus software to identify and analyse their programmes. By using non-alphanumeric programming, they might obfuscate the intent of their code and make it harder for static analysis and signature-based detection techniques to identify and prevent malicious software. In addition, non-alphanumeric code is effective against automated systems that depend on recognized patterns to detect potential threats. As a result, non-alphanumeric code has become a popular weapon in the arsenal of malware

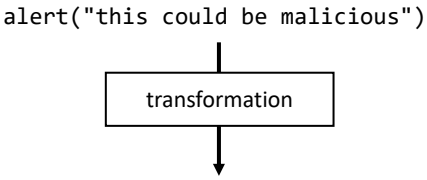
developers, enabling them to design more complex and elusive threats that may elude detection and continue to wreak havoc on computer systems without being discovered.

Creating non-alphanumeric code by hand or creating tools for automating the process is challenging, so the malware authors use or modify existing tools to perform this technique. One such tool is a Non-Alphanumeric JS obfuscator which converts any JavaScript code into a code that contains only these characters:

_	-	,	;	:	!	.	(	)	[
]	{	}	*	/	+	=	~	$	

In Figure 17, a simple JavaScript example code is converted to non-alphanumeric JavaScript code while keeping the same functionality. The generated non-alphanumeric code appears to be impossible to reverse engineer. However, automated reverse-engineering methods can be used to obtain the original code immediately.

This kind of obfuscation has high potency, meaning that it is difficult for a human to understand, but it has low resilience since it is easy to revert and moderate cost since generated source code is much larger than the original, which also means that the code is slower to execute.




```
_=~[];_={_:{_:!![]+[],$_:![]+[],$:[]+{},__:[][_]+[]},_$:++_,$_:++_,$$:-~_++,$:-~_,__:-
~++_},_._.$$_=_._.$[_.__+_.$_]+_._.$[_.$_]+_._.__[_.$_]+_._.$[_.$]+_._._[_._$]+_._._[_.$_]+_._.__[_._
$]+_._.$[_.__+_.$_]+_._._[_._$]+_._.$[_.$_]+_._._[_.$_],_.$$_=[][_._.$$_][_._.$$_],_._._=[]+/[]/[_._
.$$_],_._.$$=[]+([]+[])[_._.$$_],_._._$=_._._[_._$]+_._.$[_.$_]+_._.$$[_.$_+_.__+_.__]+_._._[_._$]+_.
_._[_.$_]+_._.__[_.$_+_.__]+_._.__[_.$_]+_._.$$[[]+_.$_+_.__];_._.$__=_._._[_._.$_[_.$$]+_._.__[_.$_+_
.__]+_._.__[_.$_]+((_.$$+_.$)*_.__)[_._.__$](_.$*(_.$+_.__))](),_._.$$$=_._._[_.$_]+_._._[_.$]+_._._[_
._$]+_._._[_.$$]+_._._[_.$_]+_._.__[_.$_];_._._$$=[]+_.$$_(_._.$$$+_._.$[_.$+_.__]+([]+~_._$)[_._$]+_.
$_+_._.$__[[]+_.$$+(_.$$+_.$)]+_.$$)(),_.$$$=_.$$_(_._.$$$+_._.$[_.$+_.__]+_._._[_.$]+_._.$_[_.$]+_._.
$[_.__+_.$_]+_._.$_[_.$_]+_._.___[[]+_.$_+_.__]+_._._[_.$])(),_.$__=_.$$_(_._.$$$+_._.$[_.$+_.__]+_._.
_[_.$$]+_._.__[_.$_]+_._._[_.$]+_._.$_[_.$]+_._.$[_.__+_.$_]+_._.$_[_.$_]+_._.___[[]+_.$_+_.__]+_._._[
_.$])();_._._$=[]+_.$$$(_._.$[_.$+_.__])[_._$];$={$_:_._.$_[_.$_],_:_._.$_[_.$$],$$:_._._[_.$],__$:_._
._[_.$_],_$:_._._[_._$],$_$:_._.$$[_.$*(_.$$+_.$)],___:_._._$__[_.__+_.__],_$_:_._.$__(_._._$+(_.$+_.$)+(
_.__+_.__)),$:_._.__[_.$_+_.__],$__:_._.$_[_.$],__:_._.$[_.$+_.__],$$_:_._.$[_.__+_._$],$$$:_._.$[_.$_
],_$$:_._._[_.$$],____:_._.__[_.$$],$___:_._.$[_.$$],$$__:_.$__(_._._$+(_.$+_.$)+_._.__[_.$$]),$$$_:_.
_.$$[_.__*_.__]};_.$$_($.$_+$._+$.$$+$.__$+$._$+$.$_$+$.___+$._$+$._$_+$.$+$.$__+$.__+$.$$_+$.$$$+$._$
$+$._+$.____+$.__+$.$___+$.$$+$.__+$.$$__+$.$_+$._+$.$+$.$$_+$.$+$.$$$+$._$$+$.$__+$.___+$.$$$_)()
```

Figure 17. Non-alphanumeric JavaScript code example

Static analysis is challenging but not impossible since it is possible to detect long blocks of meaningless code (with none or a low amount of recognized instructions) that might indicate obfuscated code. However, secure programmes often use obfuscated code for security concerns, which might lead to false positives. By analysing the code structure, variable names, function calls, and other grammar features, static analysis tools may discover obfuscation and reveal trends for further research, such as substituting letters with symbols or numbers.

Dynamic analysis techniques may identify non-alphanumeric obfuscation by observing the execution of the code and searching for anomalous behaviour or activity patterns that may suggest obfuscation. For instance, a dynamic analysis tool may identify that a programme is creating strange or unexpected strings during execution, which might suggest obfuscation.

3.9 Polymorphism

Polymorphism is a widely used term in cyber security to describe the ability of malware to alter its characteristics and evade security measures. Table 12 displays classification information for the technique. The fact that the term "polymorphism" means "many forms" suggests that the malware is capable of assuming various shapes or characteristics. By masquerading as a legitimate system process or service, polymorphic malware is designed to evade detection systems. According to Ann Johnson, 96 per cent of Windows Defender's detections in 2017 were polymorphic malware. The use of polymorphic malware by cybercriminals is on the rise. However, polymorphism is not exclusive to cyber security and is an essential programming concept. Polymorphism is the foundation of Object-Oriented Programming in programming. (OOP). It provides a single interface to multiple entity classes, allowing multiple entities to be represented by a single symbol. Without polymorphism, communication between applications would be unthinkable. Polymorphism is, therefore, a foundational concept in computer science, and its applications are not limited to cyber security. (Unterfingher, 2021)

Table 12. Polymorphism technique classification

Technique	**Polymorphism**
Aliases	Oligomorphic, polimorphic and metamorphic obfuscation
Categories	Behavioural Obfuscation Techniques Code Modification Techniques Code Transformation Techniques Control Flow Obfuscation Techniques Encoding and Encryption Techniques
Potency, resilience and cost	
Potency	High
Resilience	High
Cost	Medium

Sophistication, detection difficulty and impact	
Sophistication	High
Detection difficulty	High
Impact	High

Most commercial antiviruses are signature-based and use existing database signatures to detect the malware. Malware authors use code obfuscation techniques in their variety of malware with the aim of bypassing detection by antiviruses. Metamorphic malware changes its internal structure hence evading signature-based detection. To address the shortcomings of simple encrypted malware, the malware authors devised technologies to change the decryptor signature from one generation to the next to make it undetectable by traditional techniques. (Lin & Stamp, 2011)

The first effort was with oligomorphic malware, which uses a new decryption routine for each infection, allowing a single source malware to generate hundreds of new strains by cycling through decryption routines. As no two infections are identical, this clever mechanism makes it difficult for anti-malware software to detect and eliminate the threat. However, this method is not foolproof, as the permutations will eventually run out and are detectable using signatures-based detection. (Unterfingher, 2021)

To overcome this limitation, malware authors developed polymorphic malware. This type of malware can generate unique code signatures by employing techniques such as randomly inserting dead code (see Figure 8 on page 52), reassigning registers (see Figure 22 on page 78), and shifting and modifying data unnecessarily. In addition, malware authors can modify their malicious code and generate undetectable code signatures using tools like “The Mutation Engine” to simplify the process. Polymorphic malware goes beyond simply cycling through decryption procedures, as it uses a mutation engine to fuse the decryption routine with the encrypted malware's body, resulting in a new strain of malware that can evade the majority of signature-based detection techniques. (Unterfingher, 2021)

The first example of advanced polymorphic malware called Win95/Zmist surfaced in 1999. This malware was a game-changer, as it could modify its code and evade detection by antivirus

programs. Unlike traditional malware that security systems would detect by its signature or behaviour, Zmist used advanced code integration technology to infiltrate the target software. It did this by breaking the target program into small, manageable blocks and infecting each block with its code. Once each block had been infected, Zmist would reassemble the original program and the malware into a single executable file. This made it incredibly difficult for antivirus programs to detect malicious code and stop the malware from spreading. The discovery of Zmist was a wake-up call for the security industry, highlighting the need for better detection and prevention systems to combat the increasingly sophisticated methods of cyber criminals. (Lin & Stamp, 2011)

Metamorphic malware is the next level up from polimorphic malware. Each generation of metamorphic malware modifies its signature by identifying, parsing, and altering its own body, as seen in Figure 18. As a result, metamorphic malware never reveals its final composition, and constantly changing code signatures make it difficult for anti-virus scanners to detect malicious code. Metamorphic malware decompiles its code into meta code (a representation of the original code) and then writes a new executable code based on the meta code. Written code is ensured to be unique using methods like code reordering, instruction and register substitution. (You & Yim, 2010)

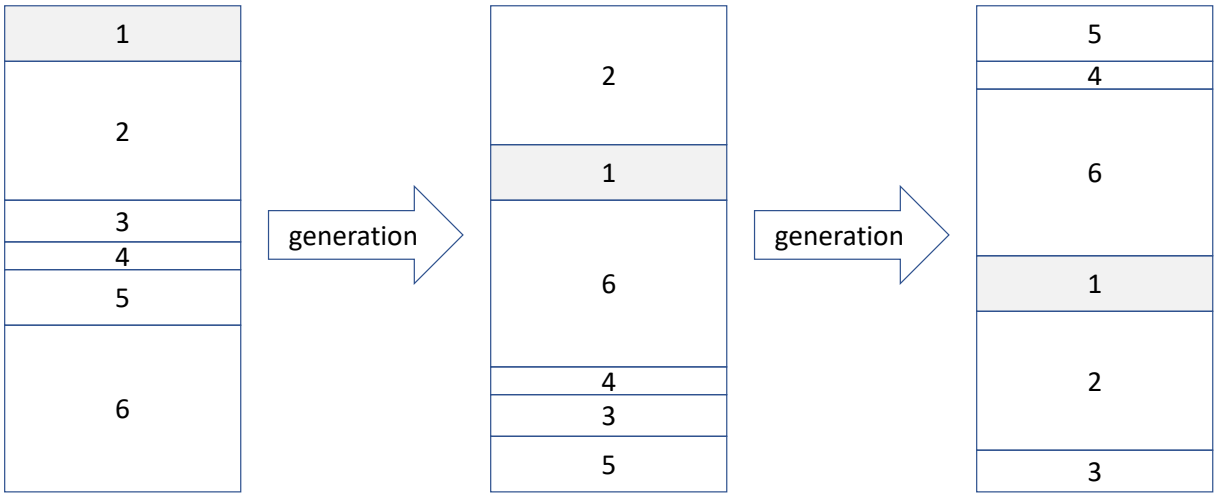


Figure 18. Metamorphic changes between generations

Unlike previous methods employing encrypted code, metamorphic malware does not require decryption. Instead, it employs a mutation engine that modifies the malware's entire source code to generate self-modifying programmes. This mutation engine applies code transformation rules iteratively to modify the syntax of the code, making it difficult for signature-based detection systems to identify. The primary components of the metamorphic malware are a disassembler, a code

transformer (obfuscator), and an assembler. First, the disassembler converts the code into assembly instructions. Next, malware's binary sequence undergoes modifications by the code transformer, which employs semantic-preserving rewriting techniques like Dead-code insertion (a), code transposition (b), and register substitution (c) as shown in Figure 19. Finally, the assembler converts the modified assembly code back into machine code. The result is a syntactically distinct but semantically equivalent programme, which we refer to as metamorphic variants. (Campion et al., 2021)

a.) Original code

```
call 0h
pop ebx
lea ecx, [ebx + 42h]
push ecx
push eax
push eax
sidt [esp – 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

b.) Original code obfuscated through dead-code insertion

```
call 0h
pop ebx
lea ecx, [ebx + 42h]
nop
nop
push ecx
push eax
inc eax
push eax
dec [esp – 0h]
dec eax
sidt [esp – 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

c.) Previous code obfuscated through code transposition

```
      call 0h
      pop ebx
      jmp J2
J5:   pop ebx
      add ebx, 1Ch
      cli
      jmp J6
J4:   push eax
      dec [esp – 0h]
      dec eax
      sidt [esp – 02h]
      jmp J5
J3:   nop
      nop
      push ecx
      push eax
      inc eax
      jmp J4
J2:   lea ecx, [ebx + 42h]
      jmp J3
J6:   mov ebp, [ebx]
```

d.) Previous code obfuscated through register substitution

```
      call 0h
      pop edx
      jmp J2
J5:   pop edx
      add edx, 1Ch
      cli
      jmp J6
J4:   push ecx
      dec [esp – 0h]
      dec ecx
      sidt [esp – 02h]
      jmp J5
J3:   nop
      nop
      push ebx
      push ecx
      inc ecx
      jmp J4
J2:   lea ebx, [edx + 42h]
      jmp J3
J6:   mov ebp, [edx]
```

Figure 19. Multiple obfuscations applied in imaginary metamorphic malware

Polimorphic malware was a significant issue until anti-virus scanners began to use methods other than signature-based detection, such as executing possible malware within an emulator (known as a “Sandbox”). When the malware was permitted to run in an emulator, the anti-virus could utilise more traditional detection methods, such as signature-based detection, to identify the now decrypted payload in memory. Alas, polimorphic malware evolved again and started detecting and defeating such emulation environments. However, in the end, as the anti-virus scanners advanced and matured, the polimorphic malware was eventually defeated and prevented. Malware that has metamorphosed is more challenging to identify and defeat. It may be feasible to detect a mutation engine employed by metamorphic malware, but after the virus has migrated into a target process, it may be hard to distinguish it from normal, benign process behaviour.

Polymorphic viruses are challenging to detect due to their ability to alter their code in various ways to evade detection. Antivirus software must employ heuristic analyses to check specific areas of the programme and emulate the programme in a sandbox to identify and capture the virus in action. During the emulation process, the virus body appears decrypted in the main memory, making it easier for the software to detect and eliminate it. A virus that undergoes metamorphosis is even more challenging to detect. The complex obfuscation techniques employed by these viruses, such as code transposition, the substitution of equivalent instruction sequences, and register reassignment, make it difficult for heuristic detection techniques to identify them. In addition, they can embed the virus code within the host program, concealing the virus's entry point and rendering detection nearly impossible. A more comprehensive analysis of malicious code based on advanced static-analysis techniques is required to overcome this difficulty. It requires inspecting the code to detect malicious patterns using structures closer to the code's semantics, as purely syntactic techniques, such as regular expression matching, is no longer adequate. (Christodorescu & Jha, 2004)

3.10 Randomization

Randomization (also referred to as randomized variable and function) is an obfuscation technique employed by JavaScript and other script-based malware to make it more difficult for security researchers to analyse the code and identify malicious behaviour. Table 13 displays classification information for the technique. This technique involves replacing meaningful names with randomly generated strings, which can be generated using a variety of random string name generators or hexadecimal encoding. Multiple randomization and other methods are frequently combined to increase the code's complexity and make it more challenging to analyse. However, while obfuscated variable and function names can make it more difficult for researchers to identify malicious behaviour, they do not provide complete protection, as sophisticated analysis techniques can bypass them. (Xu et al., 2012)

Table 13. Randomization technique classification

Technique	**Randomization**
Aliases	Randomized variable and function names
Categories	Code Modification Techniques
Potency, resilience and cost	
Potency	Low
Resilience	Medium
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	Low
Detection difficulty	Medium
Impact	Low

In order to apply this technique, the code goes through a manual or automatic randomization phase where all of the variable and function names in the script are randomized. For instance, instead of naming variables in a way that identifies their purpose, such as *maliciousUrl*, a random name could be generated, such as *v43f8d9cde9a9c*. The random name is unintelligible to humans,

but it still serves a purpose within the code and does not affect how the code operates. This technique is illustrated in Figure 20, where the original JavaScript code (a) is obscured by randomising variable and function names (b).

```
function show_alert(text) {
    alert(text);
}
var hello = 'This could be malicious';
show_alert(hello);
```

(a)

```
function v29fa90d9a0(v8323a39) {
    alert(v8323a39);
}
var v093ac981f = 'This could be malicious';
v29fa90d9a0(v093ac981f);
```

(b)

Figure 20. Randomized variable and function names

Static or dynamic analysis is mostly unaffected by this technique since the malicious code is visible and detectable. However, to defend against static signature-based detections, malicious actors can easily create multiple versions of the same malicious code by randomly renaming variables and functions. Additionally, this technique makes it more difficult for research analysts to comprehend what the code does, as they must first determine the meaning of each variable and function. This process is time-consuming but can be sped up by using automated AI and ML-powered tools. Other ways of mitigating this technique include for example pattern recognition technique that compares the code against malicious patterns of malicious JavaScript keywords (which cannot be obfuscated by this technique) and dynamic analysis using behavioural analysis.

3.11 Register randomisation

Register randomisation (or assignment) is another straightforward approach for changing program signature while maintaining functionality. Table 14 displays classification information for the technique. Registers are high-speed storage units contained within the CPU (Computer Processing Unit or Processor). Registers are visible only at the machine language level after the program is compiled into executable form. When a processor executes machine language instructions, it uses input and output registers set by the programmer to fetch and store data. In this technique, previously utilised registers are switched to other available registers while keeping the program functionality unaffected. CPU typically contains at least eight general-purpose registers that are free to be selected as input and output targets.

Table 14. Register randomization technique classification

Technique	**Register randomisation**
Aliases	Register assignment
Categories	Code Modification Techniques
Potency, resilience and cost	
Potency	Low
Resilience	Low
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	Low
Detection difficulty	Low
Impact	Low

Malware developers utilize this technique by randomizing the registers used during the program execution, consequently changing the malware's signature. By randomizing registers that are used in malicious programs, it is possible to generate an almost infinite amount of different signatures for the same malware.

The original non-obfuscated example code is shown in Figure 21, and the version using the register randomisation technique is in Figure 22. In the example, the new version uses *EDX* register to compute the same total as the *EBX* registers in the previous generation. Using different registers means the CPU uses slightly different instructions to perform the function. (You & Yim, 2010)

```
0000000140001000 <.text>:
   140001000:   55                      push   rbp
   140001001:   48 89 e5                mov    rbp,rsp
   140001004:   48 83 ec 20             sub    rsp,0x20
   140001008:   b8 2a 00 00 00          mov    eax,0x2a
   14000100d:   bb 0a 00 00 00          mov    ebx,0xa
   140001012:   29 d8                   sub    eax,ebx
   140001014:   bb 18 00 00 00          mov    ebx,0x18
   140001019:   29 d8                   sub    eax,ebx
   14000101b:   e8 00 00 00 00          call   0x140001020
```

Figure 21. Original assembly code

```
0000000140001000 <.text>:
   140001000:   55                      push   rbp
   140001001:   48 89 e5                mov    rbp,rsp
   140001004:   48 83 ec 20             sub    rsp,0x20
   140001008:   b8 2a 00 00 00          mov    eax,0x2a
   14000100d:   ba 0a 00 00 00          mov    edx,0xa
   140001012:   29 d0                   sub    eax,edx
   140001014:   ba 18 00 00 00          mov    edx,0x18
   140001019:   29 d0                   sub    eax,edx
   14000101b:   e8 00 00 00 00          call   0x140001020
```

Figure 22. Register-randomised assembly code

There are several methods for detecting and mitigating register randomization, but the static analysis is ineffective against register randomization. Using virtualization or sandboxing techniques to analyse the behaviour of malware is a practical approach. Regardless of whether register randomization is employed, it is possible to monitor malware's actions and detect any malicious ones using behaviour-based analysis. In addition, security software can use machine learning algorithms to analyse the behaviour of known malware and develop models that can detect and identify new malware variants, including those that use register randomization.

3.12 Return-Oriented Programming

Return-oriented Programming (ROP) is a technique adversaries uses to create malicious programmes that utilise pre-existing sequences of machine code, known as "gadgets," extracted from a target programme or system. Table 15 displays classification information for the technique. By interconnecting these gadgets, adversaries can execute unauthorised operations and seize control of the system. This technique is especially effective at circumventing security measures such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR), designed to prevent conventional code injection attacks. Unfortunately, ROP attacks are challenging to detect and prevent, making them a popular tool among cybercriminals with advanced skills and state-sponsored hackers.

Table 15. Return-Oriented Programming technique classification

Technique	**Return-Oriented Programming**
Categories	Behavioural Obfuscation Techniques Control Flow Obfuscation Techniques
Potency, resilience and cost	
Potency	High
Resilience	Medium
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	High
Detection difficulty	High
Impact	High

In recent years, buffer overflows have been one of the most prevalent sources of software vulnerabilities. For example, a newly found vulnerability in the OpenSSL library was caused by the user's input not being validated for length. When a buffer overflow happens, the original program's operation may be altered to begin running the code provided by the attacker.

Because buffer overflow is a common source of vulnerabilities, an operating system-level security measure was introduced known as Data Execution Prevention (DEP). This security measure allows specifying memory ranges (data blocks) critical to the application so that those memory ranges cannot be accidentally or intentionally overwritten and used for executing possibly malicious code. Even so, this security measure does not entirely solve the issue, and cybercriminals have found ways to circumvent it. (Prandini & Ramilli, 2012)

Hovav Shacham presented a technique, later named as Return-Oriented Programming approach, in 2007 when he demonstrated how to exploit concise pieces of the target program's existing code (called gadgets) effectively to run code suited for an attacker. (Shacham, 2007). Return-Oriented Programming got its name from the RETN assembly instruction, which returns the program control flow back to the address that was called the execute program block previously.

With this technique, a malicious actor or malware exploits a previously identified buffer overflow vulnerability to write malicious return addresses that point to some suitable existing short code piece (called a gadget) addresses to memory locations slavishly utilised by the original program flow. Overwriting the address that the RETN instruction utilises, the program flow effectively changes and gives complete control of the target program. By chaining multiple short pieces of code, the attacker can execute complex exploits. (Prandini & Ramilli, 2012)

Malware actors employ this technique frequently to attempt to create a reverse shell connection between the attacker and the target server. Using this method, the attacker can also elevate their privileges on the target system, as the application under exploitation may have elevated privileges, allowing the attacker to execute commands of his choosing with this new level of authority. Although operating systems are constantly improved to prevent the execution of such techniques, they are ineffective since attackers always find new ways to overcome these protections. The most effective strategy to avoid this sort of technology is for application developers to take responsibility because the primary enabler of these techniques is irresponsible programming mistakes connected to buffer overflows. It is also important to keep software and operating system constantly updated to minimize attack surface for malware attacks and configure restricted account privileges allowing only the minimum privileges necessary for the applications to perform their functions.

3.13 Self-modifying code

Self-modifying code (SMC) is an obfuscation technique that involves adding, modifying and removing instructions during execution. Table 16 displays classification information for the technique. Self-modifying code can be used in legitimate software to optimize performance, reduce the size of the code and protect software from hackers or piracy. However, self-modifying code can also introduce security vulnerabilities if not implemented correctly, making the code unpredictable and challenging to analyze. Consequently, malware and other harmful software often use this technique to conceal their true intents and evade detection by antivirus and antimalware software. Self-modifying code is also a practical approach for evading signature-based detection, in which security software looks for specific code patterns to identify malware. However, malware may circumvent detection by these signature-based approaches by continually modifying its underlying code. (Banescu & Pretschner, 2018)

Table 16. Self-modifying code technique classification

Technique	**Self-modifying code**
Categories	Behavioural Obfuscation Techniques Code Modification Techniques Code Transformation Techniques Control Flow Obfuscation Techniques
Potency, resilience and cost	
Potency	High
Resilience	Medium
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	High
Detection difficulty	High
Impact	High

To defend against self-modifying code, operating systems have introduced a memory protection feature called Data Execution Protection (DEP), commonly known as *W-xor-X*. DEP is a memory security feature that divides memory as either writable or executable, but not both, to guard against self-modifying code. This helps against self-modifying code and other memory-based attacks since executable code cannot be inserted into data areas or executed from data regions. In addition, DEP generates an access violation exception whenever an attempt is made to run code from memory locations that have been marked as non-executable using hardware support. Along with other security measures, such as Address Space Layout Randomization (ASLR), DEP is extensively employed in Windows and Linux operating systems to offer a complete defence against memory-based attacks. (Marpaung et al., 2012)

Static analysis is not an effective technique for detecting malware that uses self-modifying code. As seen in Figure 23, when self-modifying code is analysed with static analysis, commonly performed by anti-virus scanners, the malicious code is not visible since the static analysis is not executing the code and thus does not see that the code is actually changing to malicious.

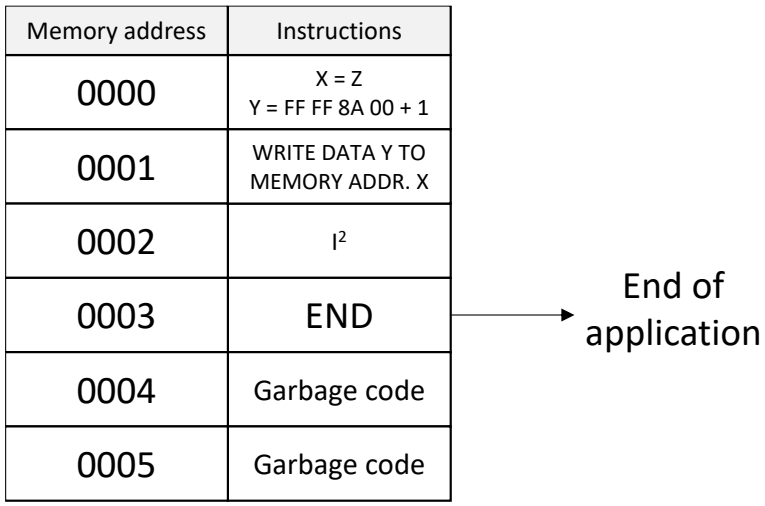


Figure 23. Self modifying code seen with static analysis

However, by using dynamic analysis to monitor the programme's behaviour as it executes, it is possible to detect the new malicious code that was added or modified by the self-modifying code. An example of code that has been changed by self-modifying code is seen in Figure 24.

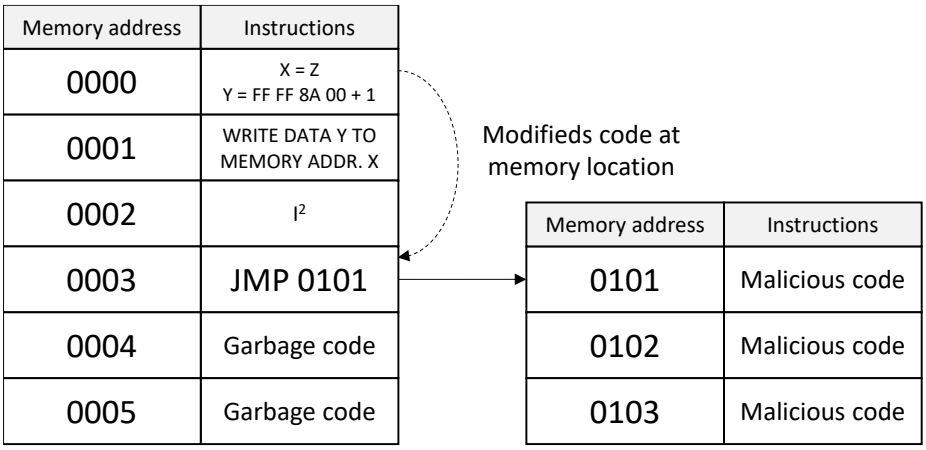


Figure 24. Self-modifying code during execution

To detect self-modifying code, dynamic analysis is required. Static analysis is ineffective since the final code is not detectable until it is written to memory during the execution of the malware. Therefore, dynamic analysis combined with behavioural-based analysis is the most effective way of detecting self-modifying code. Regular programs rarely need self-modifying code, so detecting self-modifying code is a good indication of a possibly malicious program. Also, Data Execution Protection (DEP) and Address Space Layout Randomization (ASLR) technologies are good at mitigating self-modifying code from executing.

3.14 String splitting

String splitting is a technique malware authors use to obfuscate their code and make it harder to understand and analyze. Table 17 displays classification information for the technique. This method involves separating a string or function name into multiple smaller fragments and then reassembling them at runtime. This technique is intended to make it more difficult for security researchers and antivirus software to detect and analyse malware. In addition, by separating strings and function names, the code becomes less readable and more complicated, making it more difficult to identify malicious actions and functions.

Table 17. String splitting technique classification

Technique	**String splitting**
Aliases	String reconstruction
Categories	Code Transformation Techniques Data Obfuscation Techniques
Potency, resilience and cost	
Potency	Medium
Resilience	Low
Cost	Medium
Sophistication, detection difficulty and impact	
Sophistication	Low
Detection difficulty	Medium
Impact	Low

A frequent application of string splitting is to obscure the names of functions that perform malicious actions, such as stealing sensitive data or seizing control of a system. By separating the function name into smaller pieces and reassembling them at runtime, malware can make it difficult for security researchers to identify and block the function. String splitting can also be used to obscure text strings containing sensitive information, such as URLs or login credentials. Again, by slicing the

string into smaller pieces and reassembling them at runtime, malware can make it more difficult for security researchers to identify the precise data being transmitted or stored by the malware. (Xu et al., 2012)

String splitting is a standard method employed by malware authors to make their code more difficult to understand and analyse. Therefore, security researchers and antivirus software developers must be aware of this technique and develop methods to detect and analyse string-splitting malware.

Figure 25 shows an example of using the string-splitting technique. The original JavaScript code (a) is obfuscated using string splitting technique (b), making the resulting obfuscated version hard to read, and it is impossible to instantly see what is going on.

```
alert('This could be malicious');
```

(a)

```
var jj = 's\')';
var by = 'rt(\'';
var dh = 's c';
var gf = ' ma';
var eu = 'oul';
var ii = 'iou';
var fg = 'd be';
var cg = 'Thi';
var ax = 'ale';
var hh = 'lic';
eval(ax + by + cg + dh + eu + fg +
  gf + hh + ii + jj);
```

(b)

Figure 25. String splitting

Static analysis using signature-based detection is ineffective against this technique since malicious actors can easily generate (with automation tools) multiple versions of the same malicious code by randomizing how the strings are split in the code. However, dynamic analysis is unaffected by this technique since the resulting strings are visible and detectable after they have been combined and are ready to be executed.

3.15 Whitespace decoding

Whitespace decoding is an obfuscation method used to conceal code in plain sight. It includes adding invisible characters such as spaces, tabs, and line breaks to a piece of code, making it more difficult to read and understand. Table 18 displays classification information for the technique. A security researcher Matt Kolisar, a DEFCON 16 speaker, unveiled this approach by describing how whitespace decoding can be used to hide JavaScript code inside benign-looking javascript code. The approach may be used for various objectives, including making it more difficult for reverse engineers to comprehend the code and concealing dangerous code in plain sight. (Kolisar, 2008)

Table 18. Whitespace decoding technique classification

Technique	**Whitespace decoding**
Categories	Whitespace Obfuscation Techniques
Potency, resilience and cost	
Potency	High
Resilience	High
Cost	High
Sophistication, detection difficulty and impact	
Sophistication	High
Detection difficulty	High
Impact	High

Steganography is the practice of concealing something in plain sight. On the most basic level, steganography is a method of concealing writing, whether made of invisible ink or microdots (Hamilton, 2003). Previously similar demonstrations were made for other programming languages in various programming competitions. There is even a Whitespace programming language using only whitespace characters (space, tab, and linefeed) as an instruction.

Malware code may be hidden in plain sight using only whitespace characters. It is possible to use, for example, a binary encoding by inserting a specific amount of whitespace characters like space

and tab, for example, at the end of javascript lines or in javascript comments, where space indicates a binary value of *0* and tab indicates a binary value of *1*. The binary numbers *0* and *1* are then easily decoded into ASCII characters (for example, binary code for *"a"* character is *10011110*), which may be used to build eventually malicious code. (Kolisar, 2008)

Example code in Figure 26 shows how malicious payload hides in plain sight on a being-looking javascript file. The code in the example performs three steps to decode the final payload. First, it reads the hidden payload from the end of the script lines and decodes the payload by interpreting tab characters as a binary value of 1 and space characters as a binary value of 0, resulting in a binary stream. Finally, it converts the resulting binary stream into malicious executable code using binary to ASCII conversion.

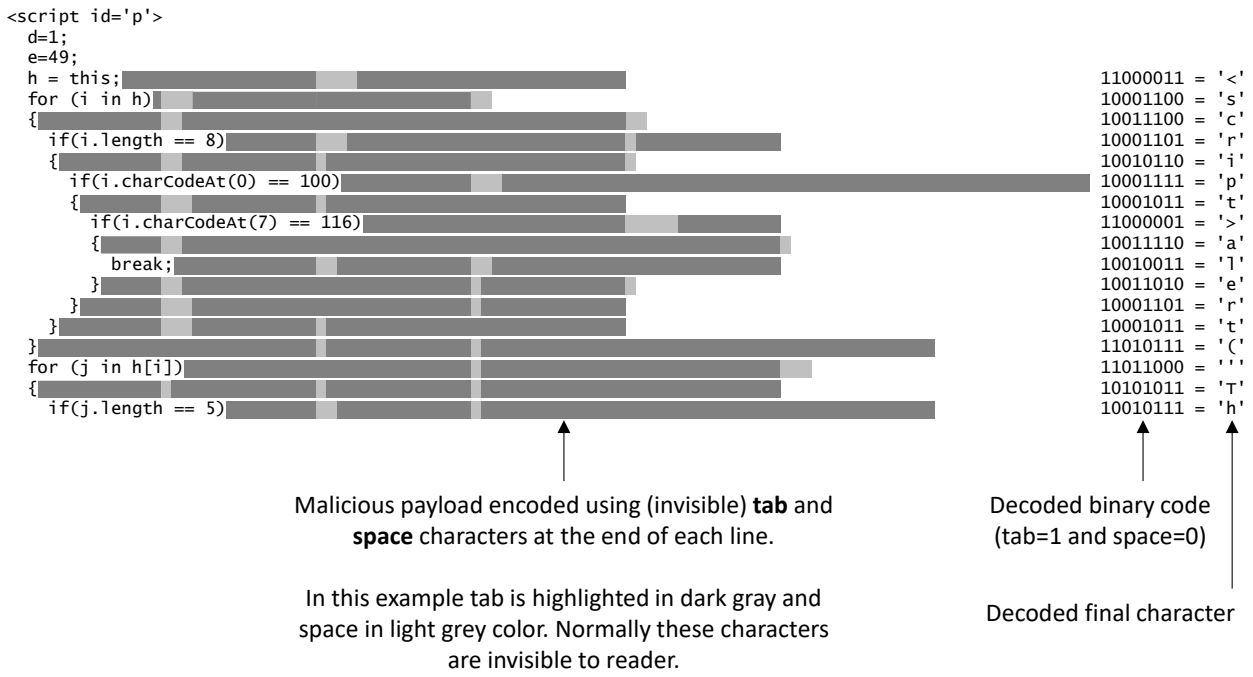


Figure 26. Whitespace encoded malicious code payload

JavaScript's whitespace encoding technique has the advantage of concealing the typical indicators of code obfuscation. Traditional methods of obfuscation result in large blocks of meaningless text using *document.write* method or differently encoded characters which are easily identifiable indicators of obfuscation. However, whitespace encoding conceals these indicators, making it more challenging for attackers to decipher the code. This technique creates a difficult-to-reverse-

engineer code and is an effective means of preventing unauthorised access to or theft of sensitive code or data.

Detecting this technique is challenging, but static analysis can detect the presence of known malicious patterns, even when encoded using this technique. Dynamic analysis is mostly unaffected by this technique, but how the code is generated and executed might present challenges. Behavioural-based detection is entirely unaffected by this technique, and malicious code is detectable as soon as it is executed.

3.16 Whitespace randomisation

Whitespace randomization is a technique used to disguise script-based malware, specifically JavaScript, by injecting random whitespace or comments. Table 19 displays classification information for the technique. These non-functional code elements do not affect the program's functionality, but they alter the code signature, making it more difficult for automatic code analysis tools, antivirus, and antimalware programmes to recognise it as known malicious code.

Table 19. Whitespace randomisation technique classification

Technique	**Whitespace decoding**
Categories	Whitespace Obfuscation Techniques
Potency, resilience and cost	
Potency	Low
Resilience	Low
Cost	Low
Sophistication, detection difficulty and impact	
Sophistication	Low
Detection difficulty	Low
Impact	Low

The method involves inserting non-functional code elements such as whitespace, comments, and semicolons. These elements are added to the code randomly while maintaining the functionality of the original code. As a result, this technique generates unique code signatures each time it is implemented, allowing malware developers to generate numerous variants of their malicious code.

In order to counteract this technique, security researchers and antivirus software developers can remove all whitespace and comment elements from the file, effectively deobfuscating the code, and then perform a signature check against known malware.

4 Conclusion

Advanced malware has become the primary weapon for cybercriminals and other cyber threat actors. The sophistication and complexity of malware attacks have increased, making them more challenging to detect and counteract. This pattern will likely continue as cybercriminals become more skilled and better equipped with cutting-edge technologies and tools. Therefore, cyber security defenders must stay ahead of this ever-changing threat landscape and adopt new and innovative methods to detect and combat advanced malware.

In recent years, *ransomware attacks* have become increasingly prevalent and are expected to continue to rise. In ransomware attacks, a hacker gains access to the victim's computer system and encrypts their files, effectively holding them hostage until a ransom is paid. *Advanced malware*, including complex viruses and spyware, is another significant threat that will increasingly cause system damage and steal sensitive data enabling threat actors to continue their attack with other attack campaigns. In the future, *social engineering threats* will also be a significant concern. In these attacks, individuals are tricked into providing access or information to a hacker. In addition, an attack may involve phishing or pretexting, in which the hacker poses as someone else to gain access and trust. *Data or internet availability threats* are also on the rise. DDoS attacks can overwhelm a system, rendering it inaccessible to users, and destruction of infrastructure can result in widespread outages and interfere with essential internet services. Misinformation and disinformation campaigns are also used to manipulate public opinion and create disorder.

As recently demonstrated by some of the most destructive malware campaigns, supply-chain attacks are becoming more sophisticated and widespread. The attacks consist of multiple phases, with the initial phase targeting a third-party vendor to compromise software or services the primary target uses. Next, intruders insert malicious code or backdoors into the vendor's software or service, which are then distributed as a legitimate software update or patch. When the compromised update is installed on the target systems, attackers can gain unauthorised access and establish a foothold within the targeted systems. The attackers then move laterally, obtaining elevated privileges and exfiltrating sensitive data, resulting in monetary losses and reputational damage for the target and its customers. This attack exemplifies the growing threat and complexity of such attacks and highlights the need for robust security measures throughout the software supply chain. Furthermore, it emphasised the importance of vigilant screening and monitoring of third-party

vendors, secure coding practices, multi-factor authentication, routine security audits, and timely patch management.

Understanding the nature and characteristics of sophisticated malware is the first step in combating it. Typically, sophisticated malware is created to evade detection by conventional antivirus and intrusion detection systems. In addition, it is frequently distributed through sophisticated social engineering techniques, including phishing emails and other forms of social engineering. Advanced malware, once installed on a target system, can steal sensitive data, spy on the victim, and even use the infected system as a launching pad for further attacks.

During the research phase of this study, it became evident that obfuscation techniques have become a persistent issue for the cyber security industry, as they make it more difficult for analysts to identify and mitigate threats. This study demonstrated that obfuscation could take many forms, including using packers or crypters to conceal the true purpose of malicious code and incorporating anti-analysis techniques that make it challenging for researchers to analyse malware samples in a secure environment. In addition, research showed that obfuscation can be used to conceal network traffic patterns, making it more challenging for defenders to detect and block malicious communications. Even though there are various tools and techniques to assist researchers and defenders in identifying and analysing obfuscated code, the ongoing arms race between attackers and defenders means that obfuscation remains a significant challenge for those working to secure our digital infrastructure.

As learned in this research, anti-virus products rely heavily on file signatures to identify malware, though most also employ heuristic detection to help identify suspicious behaviour. Malware developers continually develop new evasion techniques, making it challenging to stay ahead of them. AI-based tools have come into play, and with the assistance of AI, security researchers can analyse vast quantities of data to identify patterns and predict future attacks. Many different ongoing research projects aim to help in this task. MetaSign, for instance, can analyse a set of metamorphic variants and generate a set of transformation rules that may have been used to create those variants. In addition, identifying patterns in their obfuscation techniques allows security researchers to stay ahead of malware developers.

A significant amount of time and resources are devoted to investigating new techniques and technologies to enhance cyber security. Unfortunately, malicious actors also use techniques invented by researchers to develop new malware variants and attacks, providing malware developers with a significant advantage.

The most crucial stage for malware is to remain undetected until the malicious execution can begin. Once malware is active on a device or network, it can begin to carry out its intended function, such as stealing sensitive data, corrupting files, or remotely controlling the victim's device. However, if the malware is detected during the transfer phase or just before it is executed, it can be quarantined and stopped by antivirus, antimalware or other end-point protections before it can cause damage.

An operating system (OS) employs multiple levels of protection, such as user access levels, which play a crucial role in preventing malware by restricting malicious software's access to specific system resources and functions. The operating system also defines rules for access levels and permissions granted to various users and processes. Cyber security experts can adjust the rules further to be more efficient in preventing malware from executing commands or accessing files beyond their granted permissions, making it difficult for malware to infiltrate the system and cause damage.

However, malware developers are constantly searching for ways to circumvent these safeguards. One way they do this is by exploiting vulnerabilities in the operating system or applications to gain elevated privileges, thereby granting them the ability to perform actions outside the scope of their authorised access. In addition, by disguising malware as a legitimate application or sending phishing emails to convince users to install malware, social engineering can trick users into granting the malware elevated privileges.

Utilizing artificial intelligence (AI) and machine learning (ML) techniques is one of the most promising strategies for detecting and mitigating advanced malware. Cyber security solutions powered by AI can analyse vast amounts of data in real time, identify anomalies, and detect previously unknown threats. Likewise, ML algorithms can learn from historical attack data to recognise patterns and predict future attacks, allowing cyber security defenders to remain one step ahead of threat actors.

However, malware developers can also employ artificial intelligence (AI) and machine learning (ML) to create more sophisticated attacks that can evade traditional security measures. In addition, malware that employs AI and ML can adapt and evolve to evade detection, making it more difficult for cyber security professionals to detect and prevent attacks. Overall, malware developers have a significant advantage when they can utilise techniques created by researchers to create new, more sophisticated attacks.

A proactive approach to threat intelligence gathering and sharing is another essential element of a successful cyber security defence strategy. Cyber security defenders must be aware of the most recent threats and trends in the malware landscape, including new variants and attack techniques. This necessitates collaboration and information sharing between diverse organisations and industry sectors to stay abreast of the most recent threats and develop effective countermeasures.

Cyber security defenders must also implement robust security controls, best practices, AI and ML techniques, and proactive threat intelligence gathering. This includes employing robust passwords, implementing two-factor authentication, and ensuring that software and operating systems have the most recent security patches. Additionally, organisations must provide their employees with regular security training to increase their awareness of cyber security threats and how to avoid falling victim to them

Lastly, cyber security defenders must have an effective incident response plan in place. For example, if a malware attack is successful, organisations must be able to quickly detect and respond to the attack, limit the damage, and restore affected systems and data. This necessitates a coordinated, well-executed response plan involving all relevant parties, including IT, security, legal, and management teams.

In conclusion, the arms race between malware and security developers will continue. While the ultimate victor is uncertain, cutting-edge technologies such as artificial intelligence and machine learning provide new tools to combat malware and stay ahead of the curve. Developing AI-based security tools and adopting proactive security measures can assist cyber security defenders in detecting and mitigating sophisticated malware attacks. Utilizing AI-based tools can also aid in creating more effective security measures, allowing security researchers to stay one step ahead of

threat actors. However, individuals must be proactive in their cyber security practices. This includes using robust passwords, updating software and security protocols frequently, and exercising caution when sharing personal information online. Additionally, to defend against upcoming threats, it is essential to remain current on the most recent cyber security threats and trends and seek credible resources and advice.

References

Abraham, S. (2017). How Antivirus Software Works (Detection Science and Mechanism). *Malware-Fox*. https://www.malwarefox.com/how-antivirus-works/

Ali, F. (2022). *Everything You Need to Know About Operation Aurora*. MUO. https://www.makeuseof.com/operation-aurora/

Arntz, P. (2021, October 28). *What is fileless malware?* Malwarebytes Labs. https://blog.malwarebytes.com/explained/2021/10/what-is-fileless-malware/

Aslan, Ö., & Yilmaz, A. A. (2021). A New Malware Classification Framework Based on Deep Learning Algorithms. *IEEE Access*, *9*, 87936–87951. https://doi.org/10.1109/ACCESS.2021.3089586

Banescu, S., & Pretschner, A. (2018). *Self-Modifying Code*. https://www.sciencedirect.com/topics/computer-science/self-modifying-code

Bashari Rad, B., Masrom, M., & Ibrahim, S. (2012). Camouflage In Malware: From Encryption To Metamorphism. *International Journal of Computer Science And Network Security (IJCSNS)*, *12*, 74–83.

Bettany, A., & Halsey, M. (2017). *Windows Virus and Malware Troubleshooting*. Apress.

Brais, H. (2015, June 17). *Compilers - What Every Programmer Should Know About Compiler Optimizations*. https://learn.microsoft.com/en-us/archive/msdn-magazine/2015/february/compilers-what-every-programmer-should-know-about-compiler-optimizations

Buckbee, M. (2015). *CryptoLocker: Everything You Need to Know*. https://www.varonis.com/blog/cryptolocker

Campion, M., Dalla Preda, M., & Giacobazzi, R. (2021). Learning metamorphic malware signatures from samples. *Journal of Computer Virology and Hacking Techniques*, *17*(3), 167–183. https://doi.org/10.1007/s11416-021-00377-z

Capano, D. E. (2021). *Throwback Attack: How NotPetya accidentally took down global shipping giant Maersk*. Industrial Cybersecurity Pulse. https://www.industrialcybersecuritypulse.com/threats-vulnerabilities/throwback-attack-how-notpetya-accidentally-took-down-global-shipping-giant-maersk/

Christodorescu, M., & Jha, S. (2004). *Static Analysis of Executables to Detect Malicious Patterns*. *12*.

Cimitile, A., Martinelli, F., Mercaldo, F., Nardone, V., & Santone, A. (2017). Formal Methods Meet Mobile Code Obfuscation Identification of Code Reordering Technique. *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 263–268. https://doi.org/10.1109/WETICE.2017.23

Cobb, C. (2004). *Cryptography for dummies* [Electronic resource]. Wiley Pub.

Cohen, G. (2021). *Throwback Attack: The AIDS Trojan unleashes ransomware on the world in 1989*. Industrial Cybersecurity Pulse. https://www.industrialcybersecuritypulse.com/facilities/throwback-attack-the-aids-trojan-unleashes-ransomware-on-the-world-in-1989/

Collberg, C., Thomborson, C., & Low, D. (1997). A Taxonomy of Obfuscating Transformations. *Http://Www.Cs.Auckland.Ac.Nz/Staff-Cgi-Bin/Mjd/CsTRcgi.Pl?Serial*.

CrowdStrike Global Threat Report | CrowdStrike. (2022). Crowdstrike.Com. https://www.crowdstrike.com/resources/reports/global-threat-report/

Dang, B., Gazet, A., Bachaalany, E., & Josse, S. (2014). *Practical Reverse Engineering: X86, X64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. John Wiley & Sons, Incorporated. http://ebookcentral.proquest.com/lib/jypoly-ebooks/detail.action?docID=1629173

Dolan, S. (2013). *mov is Turing-complete*. Computer Laboratory, University of Cambridge. https://drwho.virtadpt.net/files/mov.pdf

Elsersy, W. F., Feizollah, A., & Anuar, N. B. (2022). The rise of obfuscated Android malware and impacts on detection methods. *PeerJ Computer Science*. https://doi.org/10.7717/peerj-cs.907

Emotet | What is Emotet Malware & How to protect yourself. (2023). Malwarebytes. https://www.malwarebytes.com/emotet

European Union Agency for Cybersecurity. (2022). *ENISA threat landscape 2022: July 2021 to July 2022.* Publications Office. https://data.europa.eu/doi/10.2824/764318

Faruk, M. J. H., Shahriar, H., Valero, M., Barsha, F. L., Sobhan, S., Khan, M. A., Whitman, M., Cuzzocreak, A., Lo, D., Rahman, A., & Wu, F. (2021). Malware Detection and Prevention using Artificial Intelligence Techniques. *2021 IEEE International Conference on Big Data (Big Data)*, 5369–5377. https://doi.org/10.1109/BigData52589.2021.9671434

Frankenfield, J. (2022). *What Is Bitcoin? How to Mine, Buy, and Use It*. Investopedia. https://www.investopedia.com/terms/b/bitcoin.asp

Fruhlinger, J. (2022a). *WannaCry explained: A perfect ransomware storm | CSO Online*. https://www.csoonline.com/article/3227906/wannacry-explained-a-perfect-ransomware-storm.html

Fruhlinger, J. (2022b, August 31). *Stuxnet explained: The first known cyberweapon*. CSO Online. https://www.csoonline.com/article/3218104/stuxnet-explained-the-first-known-cyberweapon.html

Haas, A., Rossberg, A., Schuff, D., Titzer, B., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. (2017). *Bringing the web up to speed with WebAssembly* (p. 200). https://doi.org/10.1145/3062341.3062363

Hamilton, D. (2003). Spies like us. *Searcher*, *11*(9), 14–19.

Heiderich, M. (2011). *Web application obfuscation* (1st edition) [Electronic resource]. Elsevier/Syngress.

Herrera, A. (2020). Optimizing Away JavaScript Obfuscation. *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 215–220. https://doi.org/10.1109/SCAM51674.2020.00029

How to detect & prevent rootkits. (n.d.). Retrieved 11 March 2023, from https://www.kaspersky.com/resource-center/definitions/what-is-rootkit

JAMK University of Applied Sciences. (2018).

Jones, C. (2021). Warnings (& Lessons) of the 2013 Target Data Breach. *Red River | Technology Decisions Aren't Black and White. Think Red.* https://redriver.com/security/target-data-breach

Kılıç, H., Katal, N. S., & Selçuk, A. A. (2019). Evasion Techniques Efficiency Over The IPS/IDS Technology. *2019 4th International Conference on Computer Science and Engineering (UBMK)*, 542–547. https://doi.org/10.1109/UBMK.2019.8907177

Kolisar. (2008). *WhiteSpace: A Different Approach to JavaScript Obfuscation*. https://defcon.org/html/links/dc-archives/dc-16-archive.html

Krigman, A. (2022). *Ukrainian Power Grid Attack - Blog*. GlobalSign. https://www.globalsign.com/en/blog/cyber-autopsy-series-ukranian-power-grid-attack-makes-history

Kurane, S. (2014). JPMorgan data breach entry point identified: NYT. *Reuters*. https://www.reuters.com/article/us-jpmorgan-cybersecurity-idUSKBN0K105R20141223

Lee, J., Chang, H., Cho, S.-J., Kim, S. B., Park, Y., & Choi, W. (2012). Integration of Software Protection Mechanisms against Reverse Engineering Attacks. *International Information Institute (Tokyo). Information*, *15*(4), 1569–1578.

Lin, D., & Stamp, M. (2011). Hunting for undetectable metamorphic viruses. *Journal in Computer Virology*, *7*(3), 201–214. https://doi.org/10.1007/s11416-010-0148-y

Liu, C., Xia, B., Yu, M., & Liu, Y. (2018). PSDEM: A Feasible De-Obfuscation Method for Malicious PowerShell Detection. *2018 IEEE Symposium on Computers and Communications (ISCC)*, 825–831. https://doi.org/10.1109/ISCC.2018.8538691

Mahajan, G., Saini, B., & Anand, S. (2019). Malware Classification Using Machine Learning Algorithms and Tools. *2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*, 1–8. https://doi.org/10.1109/ICACCP.2019.8882965

Malwarebytes 2022 Threat Review. (2022). https://www.malwarebytes.com/resources/malwarebytes-threat-review-2022/index.html

Marpaung, J. A. P., Sain, M., & Lee, H.-J. (2012). Survey on malware evasion techniques: State of the art and challenges. *2012 14th International Conference on Advanced Communication Technology (ICACT)*, 744–749.

Mercante, A. (2018). A Brief History of Computer Viruses from Mischief to Ransomware. *Ceros Inspire: Create, Share, Inspire*. https://www.ceros.com/inspire/originals/computer-virus-history/

MITRE ATT&CK®. (2023). https://attack.mitre.org/

MITRE ATT&CK Framework: All You Ever Wanted To Know. (2022, February 28). https://www.threatintelligence.com/blog/mitre-attack-framework

Mobile Security Index | Verizon. (2022). https://www.verizon.com/business/resources/reports/mobile-security-index/

Mohanta, A., & Saldanha, A. (2020). *Malware analysis and detection engineering: a comprehensive approach to detect and analyze modern malware* (1st ed). Apress.

Mumtaz, Z., Afzal, M., Iqbal, W., Aman, W., & Iltaf, N. (2021). Enhanced Metamorphic Techniques-A Case Study Against Havex Malware. *IEEE Access*, *9*, 112069–112080. https://doi.org/10.1109/ACCESS.2021.3102073

O'Kane, P., Sezer, S., & McLaughlin, K. (2011). Obfuscation: The Hidden Malware. *IEEE Security & Privacy*, *9*(5), 41–47. https://doi.org/10.1109/MSP.2011.98

Ostroff, C., & Vigna, P. (2020). *Why Hackers Use Bitcoin and Why It Is So Difficult to Trace*. WSJ. https://www.wsj.com/articles/why-hackers-use-bitcoin-and-why-it-is-so-difficult-to-trace-11594931595

Packed Malware. (2020, August 1). https://www.arridae.com/blogs/Packed-Malware.php

Paganini, P. (2021). *SolarWinds hack: the mystery of one of the biggest cyberattacks ever*. Cybernews. https://cybernews.com/security/solarwinds-hack-the-mystery-of-one-of-the-biggest-cyberattacks-ever/

Park, H., Jung, W., & Moon, S.-M. (2015). Javascript ahead-of-time compilation for embedded web platform. *2015 13th IEEE Symposium on Embedded Systems For Real-Time Multimedia (ESTIMedia)*, 1–9. https://doi.org/10.1109/ESTIMedia.2015.7351768

Paskoski, N. (2022, February 16). *What are Double and Triple Extortion Ransomware Attacks*. RH-ISAC. https://rhisac.org/ransomware/ransomware-double-and-triple-extortion/

Prandini, M., & Ramilli, M. (2012). Return-Oriented Programming. *IEEE Security & Privacy*, *10*(6), 84–87. https://doi.org/10.1109/MSP.2012.152

Project Reporting Instructions. (2022). https://oppimateriaalit.jamk.fi/projectreportinginstructions/

Romano, A., Lehmann, D., Pradel, M., & Wang, W. (2022). Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. *2022 IEEE Symposium on Security and Privacy (SP)*, 1574–1589. https://doi.org/10.1109/SP46214.2022.9833626

Saengphaibul, V. (2022, March 15). *A Brief History of The Evolution of Malware | FortiGuard Labs*. Fortinet Blog. https://www.fortinet.com/blog/threat-research/evolution-of-malware

Schwartz, M. (2001). *Reverse-Engineering | Computerworld*. https://www.computerworld.com/article/2585652/reverse-engineering.html

Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 552–561. https://doi.org/10.1145/1315245.1315313

Sikorski, M., & Honig, A. (2012). *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press.

Singh, J., & Singh, J. (2018). Challenge of Malware Analysis: Malware obfuscation Techniques. *International Journal of Information Security Science*, *7*(3), Article 3.

Sophos Threat Report. (2022). SOPHOS. https://www.sophos.com/en-us/labs/security-threat-report

The Carbanak hacker group stole $1 billion USD. (2015). https://www.kaspersky.com/blog/billion-dollar-apt-carbanak/7519/

Tripathy, S. N., Das, S. K., Mishra, B. K., & Samantray, O. P. (2018). A Study on Malware Taxonomy and Malware Detection Techniques. *International Journal of Engineering Research & Technology*, *3*(16). https://doi.org/10.17577/IJERTCONV3IS16130

Unterfingher, V. (2021, June 25). Malware Polymorphism. Polymorphic vs. Oligomorphic vs. Metamorphic Malware. *Heimdal Security Blog*. https://heimdalsecurity.com/blog/polymorphic-malware/

Wang, X., Zhang, Y., Zhao, L., & Chen, X. (2017). Dead Code Detection Method Based on Program Slicing. *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 155–158. https://doi.org/10.1109/CyberC.2017.69

What are Computer Viruses? | Definition & Types of Viruses. (n.d.). Fortinet. Retrieved 11 March 2023, from https://www.fortinet.com/resources/cyberglossary/computer-virus

What are Petya and NotPetya Ransomware? (2022a). Malwarebytes. https://www.malwarebytes.com/petya-and-notpetya

What are the different types of malware? (2021). Www.Kaspersky.Com. https://www.kaspersky.com/resource-center/threats/types-of-malware

What is a Computer Worm? (n.d.). Malwarebytes. Retrieved 11 March 2023, from https://www.malwarebytes.com/computer-worm

What is a Trojan horse and what damage can it do? (2023, March 3). Www.Kaspersky.Com. https://www.kaspersky.com/resource-center/threats/trojans

What is Adware? – Definition and Explanation. (2022, May 11). Www.Kaspersky.Com. https://www.kaspersky.com/resource-center/threats/adware

What is malware? Definition and how to tell if you're infected. (2022b). Malwarebytes. https://www.malwarebytes.com/malware

What Is Ransomware? | Trellix. (n.d.). Retrieved 8 March 2023, from https://www.trellix.com/en-us/security-awareness/ransomware/what-is-ransomware.html

What is Spyware? (2022, May 4). Www.Kaspersky.Com. https://www.kaspersky.com/resource-center/threats/spyware

What is the CIA Triad and Why is it important? (2022). Fortinet. https://www.fortinet.com/resources/cyberglossary/cia-triad

Xu, W., Zhang, F., & Zhu, S. (2012). *The power of obfuscation techniques in malicious JavaScript code: A measurement study* (p. 16). https://doi.org/10.1109/MALWARE.2012.6461002

You, I., & Yim, K. (2010). Malware Obfuscation Techniques: A Brief Survey. *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, 297–300. https://doi.org/10.1109/BWCCA.2010.85

Appendices

Appendix 1. Obfuscation techniques in MITRE ATT&CK MATRIX