Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Erlend Frankrig

Automated Binary Differential Analysis and Behavior Identification for Closed-Source Software Supply Chain Attack Detection

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
Co-supervisor: Felix Leder
June 2024

NTNU
Norwegian University of Science and Technology

Erlend Frankrig

Automated Binary Differential Analysis and Behavior Identification for Closed-Source Software Supply Chain Attack Detection

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
Co-supervisor: Felix Leder
June 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Automated Binary Differential Analysis and Behavior Identification for Closed-Source Software Supply Chain Attack Detection

Erlend Frankrig

2024/06/03

Abstract

The continuous development and expansion of applications increases the complexity of software supply chains. Threat actors leverage these supply chains and the trust between suppliers and customers to compromise suppliers and target their customers and users. By inserting malicious code into benign software and distributing it through benign updates or installers, the attack can be challenging to detect. In this project we present an automated approach, using existing tools, to identify behavior and capabilities in software updates and generate a malicious score based on these features. We also determine how the identified behaviors can be used with machine learning methods for classification. Results show that we can identify the new and modified functions between software versions in benign and malicious updates, using binary differentiation, and identify the behaviors and capabilities in these functions. We compared the behaviors identified in benign software updates to those found in malicious updates and propose a set of behaviors and capabilities that on average have a higher prevalence in malicious updates than benign. The behaviors and capabilities are mapped to the standardized formats of MITRE ATT&CK® techniques and Malware Behavior Catalog (MBC) identifiers, presenting an advantage in further interoperability and reporting. Classification showed relatively low detection rates but also low false positives. Thus, presenting a possible addition to existing malware detection but not applicable as a primary detection method. This research covers the implementation and performance of behavior identification in software updates and the evaluation of the identified behaviors as attributes for detecting closed-source software supply chain attacks.

Sammendrag

Den kontinuerlige utviklingen av applikasjoner øker kompleksiteten i programvare-leverandørkjeder. Trusselaktører utnytter disse leverandørkjedene og tilliten mellom bruker og leverandør for å kompromittere kunder gjennom leverandørene. Ved å introdusere skadelig kode i legitim programvare og distribuere det gjennom vanlige oppdateringer eller installasjonsfiler, kan angrepet være utfordrende å oppdage. I denne oppgaven presenterer vi en automatisert tilnærming, ved bruk av eksiterende verktøy, for å identifisere oppførsel og kapabilitet i programvare-oppdateringer og genererer en verdi på hvor skadelig oppdateringen er, basert på oppførselen. Vi viser også hvordan oppførsel can brukes med maskinlæringsmetoder for klassifisering. Resultatene viser at tilnærmingen klarer å identifisere nye og endrede funksjoner mellom programvareoppdateringer, i legitime og skadelige oppdateringer, ved bruk av binær differensiering, samt identifisere oppførsel og kapabilitet i disse funksjonene. Vi sammenliknet oppførselen identifisert i legitime og skadelige programvareoppdateringer, og presenterer et utvalg oppførsler og kapabiliteter som forekommer oftere i skadelige oppdateringer. Oppførsel og kapabilteter kobles til de standardiserte formatene til MITRE ATT&CK og Malware Behavior Catalog (MBC), som kan være en fordel ved integrasjon av vår tilnærming i andre rammeverk og rapportering. Klassifisering resulterte i lave deteksjonsrater men også lave antall falske positive. Noe som tilsier at metoden har potensiale for å utvide eksisterende deteksjonsmetoder, men ikke vil være effektiv som en primærmetode for deteksjon av skadelige oppdateringer. Denne oppgaven tar for seg implementasjon og utførelse av tilnærmingen for å identifisere oppførsel i programvareoppdateringer og evalueringen av disse som attributter for deteksjon av skadelige oppdateringer i leverandørkjedeangrep.

Acknowledgement

This project represents the final subject and my master thesis for the Experience-based Master in Information Security at the Norwegian University of Science and Technology (NTNU), Faculty of Information Technology and Electrical Engineering.

I would like to thank my supervisor Dr Geir Olav Dyrkolbotn at NTNU and co-supervisor Dr Felix Leder at Crosspoint Labs, for providing support and guidance throughout this project and keeping me on the right track. I would also like to thank the rest of the research group for great discussions and input. Finally, I would like to thank my partner, Elin, for the support and patience, making it possible for me to complete this thesis.

Contents

Figures

Tables

Code Listings

Acronyms

API Application Programming Interface. 10, 17

ASCII American Standard Code for Information Interchange. 8

C2 Command-and-control. 8, 14, 15

CFG Control Flow Graph. 16

CPU Central Processing Unit. 8

DLL Dynamic Linked Library. 12, 14, 15, 29, 42, 44

HTTP Hypertext Transfer Protocol. 8, 47, 49

IAT Import Address Table. 8

ICS Industrial Control Systems. 14

IP Internet Protocol. 8, 17, 47

MBC Malware Behavior Catalog. xiii, xv, 17, 18, 23–26, 31, 34, 35, 37, 40, 41, 43, 44, 47, 49, 51, 83

MLP Multi-Layered Perceptron. 25, 26, 40, 48

SSCA Software Supply Chain Attack. xv, 1–3, 11, 12, 23, 24, 27, 28, 31, 36, 45, 49, 50, 52

TTP Tactics, Techniques, and Procedures. 17

URL Uniform Resource Locator. 8

WMI Windows Management Instrumentation. 34, 35, 43, 49

Chapter 1

Introduction

1.1 Topics covered

A significant part of our lives involves the use of software and applications. We use applications on our phones, smartwatches, and computers to communicate with friends and family, track activity, administer finances, and much more. Governments, businesses, and organizations are highly dependent on a wide range of software to conduct business and operations successfully. Most applications have several external dependencies to other applications created by third parties. These chains of application dependencies are part of the software supply chain and can become large and complex. The increase in software and the continuous development of existing software to expand functionality further increases the complexity. The attack surface increases along with the software supply chain because each software component or the supplier themselves can have vulnerabilities or weaknesses that can be exploited by attackers [1]. Thus, making it more difficult to maintain control of the attack surface, which is critical in defending against cyber threats.

Attacks on software supply chains have impacted many companies and are estimated to cause increased costs over the next years [1]. Software supply chain attacks (SSCA) have provided nation-state actors with access to critical infrastructure and enterprise networks in several sectors, facilitating for disruption and espionage [2]. In a software supply chain attack, a software supplier is attacked with the intent of further compromising their customers or users, taking advantage of the trust established between them [1]. Attackers can exploit vulnerabilities in the software supply chain to stage an attack on a target, or they can compromise the supplier and insert malicious code in their software which compromises users [2].

The attacks on SolarWinds [3], CCleaner [4], and M.E.Doc [5] show how state actors have successfully performed sophisticated SSCAs through update hijacking of closed-source software [2]. In update hijacking, the threat actor gains access to the software development or distribution environment and inserts malicious code in the software, delivering malware with a legitimate program update or installer, often with valid code signatures [1, 2].

To reduce the threat from software supply chain vulnerabilities, the cyber security community and industry use methods to identify vulnerabilities in software [6]. Binary code differential analysis is used to find code differences between software updates, decreasing the amount of code needed to be analyzed for vulnerabilities. This is also used by malware analysts to find similarities and differences in malware samples, used for classification and tracking malware development [7]. To further reduce the workload on researchers, automated malware analysis tools have also been created to identify malicious behaviors and capabilities in software [8]. Thus, these techniques are relevant for application in analyzing closed-source software supply chain attacks.

1.2 Keywords

Closed-source software supply chain attacks, trojanized updates, binary differentiation, malware behavior, malware detection.

1.3 Problem description

Research on software supply chain attacks has mainly focused on open-source software, including the detection of vulnerabilities and malicious code in open-source repositories and package managers [9, 10]. Despite the demonstrated impact and threat posed by closed-source SSCAs, the amount of research on this topic is low. However, recent research on closed-source software supply chains has contributed knowledge and approaches to improve defences against such attacks. The methods rely on finding malicious indicators in a benign and a malicious version of closed-source SSCAs using basic static and dynamic tools and techniques and conducting differential analysis on the results [11, 12]. They require the setup of several dynamic and static analysis tools and the knowledge to interpret the results from each tool. Dynamic analysis can be difficult, particularly when executing software components that are part of a larger application, as they may require certain settings or other dependencies to run.

One method has been proposed with a proof-of-concept to detect trojanized binaries in software supply chains based on general malware indicators. Validation of the methods is challenging due to the low number of known closed-source software supply chain attacks, however, comparing them to indicators found in a larger set of benign updates could provide knowledge about which indicators are useful for detection.

Furthermore, reporting the malicious indicators in a standardized format could present an improvement to existing methods by facilitating integration with existing defence methods and threat reporting.

Andreoli et al. [13] manually identified advanced static features by reverse engineering the malicious functions in closed-source SSCA and used these features' presence to classify other types of malware successfully. This research shows that the malicious code in SSCAs share at least some malicious behaviors with other types of malware. Thus, existing malware behavior identifiers may be applicable to SSCAs. However, the method of finding the malicious functions was based on existing reports and cannot be leveraged for identifying malicious functions in undisclosed SSCAs.

Based on the existing research on closed-source SSCA and the existence of automated analysis tools and techniques that are easily integrated with existing workflow, we believe there are unexplored methods of detecting trojanized software updates and update hijacking in closed-source SSCAs.

In this project, we apply automated binary code differential analysis and automated behavior identification from static analysis to compare malicious behavior and capabilities between benign and malicious software updates. The aim is to contribute more knowledge on behavior in closed-source SSCAs and develop a method for identifying malicious behavior in software updates to detect update hijacking in closed-source SSCAs, reducing the workload on malware analysts and cyber security researchers.

1.4 Justification, motivation, and benefits

Closed-source software supply chain attacks using update hijacking are not the most common type of cyber attacks but have been successfully used by nation-states in sophisticated cyber operations [2]. The potential impact and how challenging they are to detect make this an important topic for research to provide knowledge on how we can defend against them.

There are few such disclosed attacks, but they are often devastating due to the ability to target offline systems or compromise many entities simultaneously. With the amount of software and underlying components existing in our digital infrastructure, manual analysis to detect malicious updates is not feasible. Therefore, it is necessary to study ways of improving the analysis efficiency and identification of malicious behavior in software updates.

Using existing methods and tools, we develop an automated approach for identifying behavior and capabilities in software updates in a standardized format. This approach is further leveraged to present behavior differences in benign updates and malicious updates from closed-source SSCAs. The lack of research in this area is an important motivational factor for writing this thesis. Generating more knowledge and data that can be further studied or applied are steps towards improved cyber security.

1.5 Research questions

The main goal of this project is to improve the overall understanding and technical ability to combat the threat posed by closed-source SSCAs. To achieve this, we propose an approach using methods from similar problems; vulnerability research and malware behavior identification. We believe that identifying behavior from new and modified code can be used to detect closed-source software supply chain attacks.

The research questions are defined as follows:

1. How can program behavior be extracted from the changes introduced in program updates?
2. Which behaviors are prominent in benign software updates and how do they differ from malicious updates?
3. To what extent can extracted behaviors be used to identify software supply chain attacks?
4. How can identified behaviors provide a malicious score that can be used to detect malicious software updates?

1.6 Scope and contributions

The main contribution of this thesis is a novel approach for automated software behavior identification in software updates by leveraging existing tools and techniques. The approach aims to present new knowledge within the domain of closed-source software supply chain attacks and how we can mitigate the threat through detection. This includes the use of binary differential analysis for finding the updated software functions in combination with software behavior identification and reporting in standardized behavior format. We also believe that our automated approach can contribute to improving the workflow and workload for analysts.

An important contribution of this project is identifying common behavior in benign software updates, which, to our knowledge, has not been done before. Therefore, we are able to compare this to malicious update behavior and determine behavior patterns that are more likely to occur in malicious and benign updates.

We also test one possible approach for calculating a malicious score and perform classification using machine learning methods to determine if we can detect update hijacking based on identified behaviors in updates.

The focus of this thesis is on the customer side of closed-source supply chain attacks where the attack vector is update hijacking. Therefore, the datasets are created with compiled binaries of closely related versions. Due to the small number of such disclosed attacks, the number of malicious binaries is low. Also, as the best-known and most advanced attacks are SolarWinds and M.E.Doc, which are written in C# using the .Net runtime, the benign software updates analyzed in this thesis are all .NET binaries. The main advantage of this is that .NET requires less processing resources during differentiation and behavior identification than samples written and compiled in C/C++. The potential downside is that we do not account for the majority of software, which are not C# .NET binaries. To increase the sample size, the malicious dataset includes eight C/C++ compiled binaries because there only were two examples of update hijacking of .NET software.

1.7 Ethical and legal considerations

This research uses methods for finding behavioral changes and additions to proprietary software but does not present information not already publicly reported or information about how functionality is implemented. For the benign dataset, the combined overall results are presented, not revealing the source software. For the malicious dataset, the identified capabilities and the function names are compared to existing analysis to evaluate the methods. The capabilities identified are based on the rules and signatures of Mandiant's Capa tool [14], designed to identify techniques associated with malicious behavior.

1.8 Thesis outline

This thesis consists of six chapters, including this introduction. Following this chapter is a background, presenting relevant previous work and theory for this thesis. Chapter 3 describes the methodology used, including experiment setup, dataset generation and usage, and the analysis process. Chapter 4 presents the results from the experiments and the findings from the analysis. Discussion of the results and findings leading up to answering the research questions are presented in chapter 5. Finally, chapter 6 presents the conclusion and future work. The appendix will include the code generated and used in this project.

Chapter 2

Background

In this background chapter we will present existing knowledge, theory, and approaches pertaining to malware analysis and software supply chain attacks.

2.1 Malware analysis

Malware analysis is the structured way of finding out how malicious software behaves, which actions the software can take, and how it can be detected and mitigated [15]. It is often divided into static and dynamic analysis, examining the malware without executing it or while executing and analyzing its interaction with the target system [15].

Zeltser [16] describes three stages of malware analysis; behavioral, code, and memory analysis. Typically, a behavioral analysis is done first to gain an overview of the sample's behavior; also called basic dynamic analysis. This stage aims to identify capabilities and characteristics by monitoring how the malware interacts on a target system or in a specific environment. The second stage is code analysis which involves advanced static and dynamic analysis of the program's code. Reverse engineering is performed in static analysis of the disassembled program to understand how the program's capabilities and behavior are implemented. Debugging can further be used to step through each code instruction to observe the low-level behavior. This second stage can be very time-consuming but allows for detailed insight of malware behavior. The last stage involves investigating how the malware uses memory. Runtime artifacts can easier be identified compared to code analysis. Using all three stages can provide an efficient way of analyzing malware by examining the program behavior from different views. The stages do not have to be performed sequentially and can be used simultaneously to complement each other [16].

There are many different techniques and tools for analysis and they can aid in different stages of analysis and for different types of malware. In the following subsections, we will cover a few techniques and tools for malware analysis to provide background for the methodology used in this thesis.

2.1.1 Static malware analysis

Static analysis can be further broken down into basic and advanced analysis.

Basic static analysis

Basic static analysis uses the information that can be found by examining the data from the file header and existing strings or byte sequences.

Examining the file header, it is possible to identify the imported libraries and functions from the Import Address Table (IAT) [15]. These imports are used by the program so that the author does not have to implement the functionality themselves. Thus, it can provide valuable information about the behavior of the malware. For example, the import of *wininet.dll* and the function *HttpSendRequestA*[1], indicates that the program can send HTTP requests, possibly for connecting to a command-and-control (C2) server, downloading data, or extracting data. However, it is also important to keep in mind that libraries can be statically linked or imported at runtime and not show up in the IAT. Static linking will include the library in the binary and it can be harder to identify the imported functions [15]. For runtime linking, the library and function names can sometimes be observed by examining strings if they are not obfuscated [15].

Extracting the human-readable strings from the program is also a basic task that can reveal information about software behavior, such as functions imported at runtime or strings used by functions [15]. Programs with network functionality must specify the destination, which could be an IP address, domain, or URL. This must be stored somewhere in the binary and can often be found as ASCII or Unicode strings [15].

Basic static analysis is a simple approach and initial stage to get an overview of the malware's purpose and functionality. However, more advanced techniques are often necessary to fully understand how the malware works.

Advanced static analysis

Advanced static analysis is the approach of analyzing the binary code, where the goal is to understand what the code does and reveal its functionality. The binary contains the machine code which represents the instructions executed by the CPU, and it can be translated into human-readable assembly code using a disassembler such as IDA Pro[2] [15]. Reverse engineering is the analysis of the assembly code and is a powerful method as it is possible to gain detailed knowledge of exactly how the program works. It also allows for finding behavior that is not necessarily shown in dynamic analysis. However, it can be a very time-consuming task when the binary, especially when the program is large and contains many functions. Thus, reverse engineering is not feasible when dealing with a large and continuous

[1] https://learn.microsoft.com/en-us/windows/win32/api/Wininet/nf-wininet-httpsendrequesta

[2] https://hex-rays.com

flow of malware samples, but rather an approach used when analyzing novel or high-priority binaries.

2.1.2 Dynamic malware analysis

Dynamic malware analysis includes approaches where the malicious software is executed and the interaction on the system is monitored and examined. Like static analysis, we have basic and advanced techniques for dynamic analysis.

Basic dynamic analysis

Basic dynamic analysis involves running the software in a controlled environment where we can capture its actions. In the controlled environment, monitoring tools run to capture network activity, process activity, file system interaction, registry interaction [15].

Basic dynamic analysis can be very valuable as it can identify very detailed information about behavior. However, the amount of information can be very large and we must know how to run the sample with the correct settings, parameters, and possibly correct interactions to trigger the different code paths and have accurate results [15]. Some programs can be challenging to run in an analysis environment because they may rely on other software or specific environment variables. Basic dynamic analysis is also prone to anti-analysis techniques, which could cause the program to behave differently when detecting it is running in an analysis environment [15].

Advanced dynamic analysis

Advanced dynamic analysis is an approach where analysts have more control over the sample execution. Instead of just running the software and hoping it will show us its behavior, we can execute the machine code instructions step by step through debugging. Thus, we can observe what each instruction does and how values in process memory are used [15]. As with advanced static, advanced dynamic analysis is powerful but time-consuming.

2.1.3 Malware detection and classification

The behavior of malware can be identified effectively by examining the system calls performed by the malicious process [17]. The system calls can be enumerated by dynamic or static analysis. Basic static analysis may not be able to find all calls due to dynamic loading of functions and obfuscation. Advanced static analysis may however find all system calls without the need to execute the malware.

Malware detection is generally based on signatures or heuristics [18]. Signature-based detection relies on finding certain patterns found in previously seen malware. These patterns can be specific byte sequences, strings, values, or artifacts found in the binary. Signature-based detection is efficient for detecting known malware but is less capable of finding new types. Heuristic-based detection looks for behavior or characteristics uncommon for benign programs but typical for malware. This has the advantage of detecting previously unseen malware, for which we do not have a signature, but is more prone to false positives.

Machine learning methods are frequently used to create malware classifiers. These classifiers are trained on large amounts of data from known malware and benign programs to create as accurate classifiers as possible. The training data consists of a defined set of attributes suitable for classification and can be based on features or patterns found in the binaries. A classifier essentially represents a function that is learned from the training data [19]. Some common classifiers include decision trees, support vector machines, naive Bayes, and artificial neural networks [19], which represent the classifier function in different ways.

Classification based on machine learning methods can help quickly detect malware based on many patterns or patterns that are challenging to distinguish in manual analysis. Thus, reducing the workload on malware analysts.

2.2 Software supply chain attacks

Most applications have several external dependencies to other applications created by third parties. External dependencies are software components that software applications use to behave properly or perform some action [20]. These components can be libraries, functions, application programming interfaces (API), or frameworks that are created by a third party and can be open-source or closed-source [20]. For example, the Windows API is used by programs to run on and interact with the Windows operating system. Dependencies can be either direct or transitive, i.e., directly used by the application or indirectly used through other dependencies [20].

These chains of application dependencies are part of the software supply chain and can become quite large and complex. The increase in software and the continuous development of existing software to expand functionality further increases the complexity. The attack surface increases along with the software supply chain because each software component or the supplier themselves can have vulnerabilities or weaknesses that can be exploited by attackers [1]. Thus, making it more difficult to maintain control of the attack surface, which is important in defending against cyber threats.

Software supply chain attacks can target all stages in a software life cycle (figure 2.1); from the design phase throughout maintenance until it is retired [21]. Typically, the development and deployment phases are compromised. Adversaries may modify the software as it is developed, either through direct access to the development servers or through compromising external dependencies [21]. Compromising the deployment stage can allow attackers to alter the software hosted on trusted distribution servers.

A supply chain attack can be defined as a compromise of a supplier that facilitates an attack on a customer [1]. The first target is a supplier delivering software

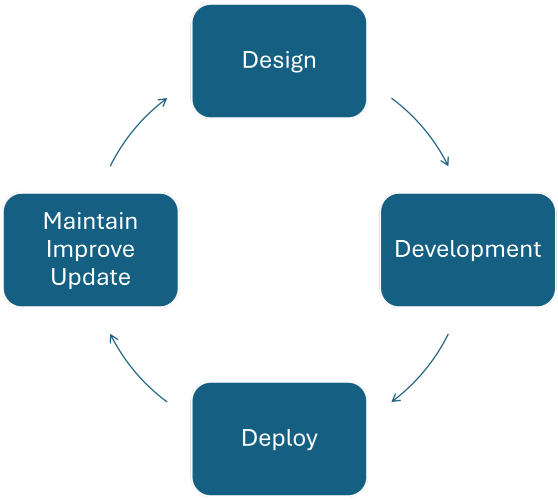

Figure 2.1: A simplified representation of the software life cycle [21]

to its customers or users, who are the intended target(s) of the attack [1]. There are several types of software supply chain attacks and they mainly differ in how the supplier is compromised.

In the following subsections, we present some types of SSCAs before we describe examples of update hijacking which is the main focus of this thesis.

2.2.1 Software supply chain vulnerabilities

Certain software vulnerabilities can allow attackers to execute malicious code on systems that use the vulnerable software. A software supply chain attack can therefore arise from vulnerabilities that exist in the software supply chain.

Log4Shell is a vulnerability that existed in the Apache Log4J library which was used by many Java applications for logging [22]. The vulnerability allowed attackers to run malicious code and gain control of the systems that used this library [22]. A few disclosed software supply chain attacks from this vulnerability include an attack on the Belgian Ministry of Defence and academic institutions in the USA [23].

This is only one example, but exploitation of software vulnerabilities is observed as one of the most used techniques for initial access in cyber attacks and it has increased in recent years [24].

2.2.2 Open-source software supply chain attacks

Uploading malicious software packages or applications to public repositories is a method leveraging the dependencies on open-source software, and is frequently observed [25]. The malicious software can masquerade as benign software by using a similar name, known as typosquatting, but including malicious code along with the original benign code [25].

Attackers can also gain access to the development process, by gaining access to developer environments or accounts or becoming a contributor to the project [2]. In the recent XZ Utils supply chain attack [26], an adversary became a contributor to the open-source project, and after a few years of gradually gaining control of the development, they added a backdoor in the software. The backdoor was included in some development versions of Linux distributions but was discovered before it was distributed to production versions of Linux and could affect millions of users [26].

2.2.3 Undermining code signing

Software is usually signed with a digital signature to assure users that the code is created and distributed by a trusted party [2]. However, attackers can undermine this process by stealing valid certificates or private keys used to sign them [2]. By signing malware with valid certificates, attackers can exploit the trust in code signing to bypass security checks and get their malware to execute on target systems [2]. The abuse of code signing does not include the delivery of malicious code but takes advantage of the trust between suppliers and customers. It can also be a part of the other SSCA types, where the malicious code is signed because it is inserted before the code-signing occurs [2].

2.2.4 Update hijacking

In update hijacking, the attacker trojanizes benign software by inserting malicious code into the benign program. The threat actor gains access to the software development or distribution environment and inserts malicious code in the software, delivering malware with a benign program update or installer, often with valid code signatures [1, 2]. If the attackers have access to the development environment, the malicious code can be included in the benign software by modifying the code and adding malicious functions. Malicious code can also be included by adding a library, such as a DLL, along with the benign program or installer.

Furthermore, threat actors can also compromise suppliers to acquire valid certificates and use this to sign a modified version of the benign program including malicious code, before distributing it from their servers [2]. Update hijacking is a widely seen method in closed-source SSCAs such as SolarWinds, CCleaner, and M.E.Doc [2, 25].

2.2.5 Detecting software supply chain attacks

A system for detecting closed-source supply chain attacks has been presented by Barr-Smith, et al. [12]. This system uses differential analysis to find malicious behavior in software builds by comparing extracted static and dynamic features between two adjacent build versions. The results show that the static features of obfuscation, packing, entropy, and Original Entry Point (OEP) changes, are the major contributors to detecting malicious behavior inserted into proprietary code.

Similar research completed by Refsnes [11], shows the use of basic static analysis and file features in differential analysis. His research shows that use of simple tools without high resource requirements or deep technical knowledge of reverse engineering can aid in the detection of trojanized binaries.

Wang et al. [27] propose a method for detecting closed-source supply chain attacks by examining command-and-control traffic during the attack. Specifically detecting exfiltration of data that is abnormal in the network. The method remains to be empirically validated but presents one possible approach to deal with supply chain attacks.

2.2.6 Examples of closed-source software supply chain attacks

The number of disclosed closed-source software supply chain attacks is low, but there are some examples using different techniques to trojanize software that are interesting to examine further.

SolarWinds

In 2020, the cyber security company FireEye discovered the SolarWinds supply chain compromise [3]. The attackers trojanized the SolarWinds Orion business software by inserting a backdoor in one of its components, leading to it being included in the build process and distributed on software updates [28]. The component including the backdoor was SolarWinds.Orion.Core.BusinessLayer.dll, a signed library loaded by the SolarWinds.BusinessLayerHost.exe, a benign executable [3]. The malicious function names and the network activity were tailored to the SolarWinds application, resembling normal and benign names and activity [3]. This may have been a reason for the backdoor not being discovered before months after distribution [28].

The SolarWinds Orion software was used by several companies and organizations worldwide, and those installing the update were compromised with the backdoor [28].

NotPetya - M.E.Doc

In 2017, attackers gained access to the development servers of the Ukrainian accounting software M.E.Doc and inserted a backdoor into one of its components *ZvitPublishedObjects.dll* [29]. The trojanized software was then pushed as software updates to infect users of the software. The backdoor provided the ability of executing commands, gather information, and deliver and execute new malware [5]. It is through this backdoor functionality that the attackers most likely deployed the destructive NotPetya malware to their targets [5]. The M.E.Doc software was used by many organizations in Ukraine and companies working there and thus the attackers were able to compromise several organizations in different sectors, such as transportation, finance, healthcare, and energy [29]. The Danish global shipping company, AP-Moller-Maersk, estimated a cost of over 200 million dollars due to the disruption of operations from the NotPetya attack [30].

Dragonfly campaign

The dragonfly campaign consists of several infection vectors, including supply chain attacks through trojanizing benign software [31]. Threat actors compromised the web servers of industrial control system (ICS) suppliers eWON, Mesa Imaging, and MB Connect Line [32]. The websites provided downloads of the suppliers' software and drivers, which the attackers changed to include backdoors [31]. The eWON software Talk2M eCatcher and eGrabit for remote access to programmable logic controller (PLC) systems were trojanized with the Havex[3] remote access tool (RAT) [32]. The Mesa Imaging driver SwissRanger for camera interfacing was trojanized with the Sysmain RAT [32].

The software was trojanized by creating a new installer, including the malware as a DLL and the original installer so the benign program would also run [32]. Thus, this campaign uses a different technique to trojanize software where the attackers did not alter the source code, but rather add malware to the installer.

SmartPSS

Mandiant [33] discovered a supply chain attack originating from the SmartPSS software provided by a security camera provider. This supply chain attack is similar to the Dragonfly campaign described above, by adding malicious functionality to the installer while executing the original legitimate software. The SmartPSS installer was trojanized by including a slightly altered legitimate windows application *mshta.exe* and modifying the installer script to execute this application with a URL as argument [33]. The URL is contacted to download a script that further downloads and executes a backdoor in memory [33].

3CXDesktopApp

In 2023 the communication software 3CX Desktop App [34] was trojanized and spread through downloads from the 3CX website [35]. The application is used by businesses and provides users with communications such as chat, video, and voice calls [36]. Mandiant [36] found that threat actors had gained access to the build environment of 3CX through an earlier supply chain attack. The 3CX Desktop App installer was trojanized by including two malicious DLLs, ffmpeg.dll and d3decompiler_47.dll [35]. The ffmpeg.dll was loaded by the application, which in turn executes the d3decompiler_47.dll and finally, contacts C2 servers to download an information stealer malware [35, 36]. Trend Micro [37] says that the DLLs were trojanzied or patched to execute the malicious functions, which indicates that the attackers had access to the software build or deployment environment.

[3] https://attack.mitre.org/software/S0093/

CCleaner

Cisco Talos [4] reported on a supply chain attack where a version of the computer cleaning software CCleaner was distributed with a backdoor. They further mention that the trojanized binary was signed with a valid certificate and included seemingly benign artifacts from the compilation. This indicated that the attackers had gained access to the development environment and modified the legitimate code to include malicious code [4]. For this supply chain attack the attackers modified a TLS callback function to call a malicious code loader before execution of the legitimate program [4]. The malicious code loads a malicious DLL which contacts C2 servers to receive instructions[4]. CCleaner is a very popular software which claims to have over 2 billion downloads worldwide and Talos' network traffic analysis showed a significant number of requests to the potential C2 domains.

2.3 Binary code differentiation

Binary code differentiation, also called binary similarity analysis, is used to determine differences and similarities in code [7]. These techniques can be used for different purposes, such as tracking changes to software, or malware, versions over time. The code changes between versions can be examined to find vulnerabilities or new functionality without the need to examine the entire program every time.

2.3.1 Methods

Similarity analysis can be divided into three categories: Syntax, semantics, and structural matching. Haq and Caballero [7] describe the methods in their binary code similarity survey. Syntax concerns the representation of data making up the objects to be compared. For binary differentiation, this could be the machine code or the assembly instructions making up the basic blocks and functions in the program. Syntax-based matching will look at similarities in these representations. Different compilers and optimizations can produce different machine and assembly instructions for the same program. Thus, syntax-based matching can fail to identify similar functions across different compilations [7].

Semantics represent the functionality of an instruction or set of instructions. Comparing semantics between binaries can therefore solve the issues with syntactic matching. However, comparing semantics for whole executable programs is too difficult and resource-heavy, but it can be possible to approximate matching by looking at the semantics of smaller parts of code [7].

Structural matching is a widely adopted approach in binary code differentiation because it is more dependable than syntax-based matching, but more computationally feasible than semantics [7]. Creating a structure of the data in each compared object and then examining the structural differences, can mitigate the problem of syntactic matching while retaining lower computational requirements than semantics. Flake [38] presents a structural method of comparing executable programs, by representing the functions of the program as graphs, called control flow graphs (CFGs). The whole executable is further represented as a call graph consisting of the relationship between the function control flow graphs. Flake describes each CFG as having only one point of entry, but may have multiple points of exit. In the CFG there are basic blocks, or nodes, which consist of assembly instructions that are grouped together by dependency and sequential execution, and split by branching instructions such as jumps [38]. Structural matching will, however, not be able to account for changes to code structure optimizations [7].

A disassembler is needed to generate the CFGs based on basic blocks and instructions. However, software written in interpreted languages, such as C# and .NET, will not be represented by assembly but rather an intermediate representation called bytecode which is translated to machine code by the interpreter at runtime[15]. The bytecode can be decompiled back to source code, not necessarily the same as the original, but often very similar [39]. The IDA Pro disassembler does not decompile the bytecode but is able to generate control flow graphs for .NET bytecode, showing how the code flow is for the software. Thus, making it possible to use structural-based matching from CFGs on .NET binaries.

2.3.2 Binary differentiation tools

Some of the practical approaches in binary code differentiation include BinDiff [40], QBinDiff [41], Ghidriff [6], DeepBinDiff [42], and Diaphora [43].

BinDiff is an open-source program for finding differences and similarities between executable files using the disassembled code [40]. It provides the ability to examine patches from vendor software, where the code is unavailable, and eases the tracking of changes to software [40]. BinDiff uses the disassembled code to generate call graphs and control flow graphs to conduct structure-based matching [38, 44]. Thus, a disassembler such as IDA Pro, Binary Ninja, or Ghidra is required to generate the disassembly code [40].

Ghidriff [6] is a tool for comparing binaries using Ghidra as a disassembler and for displaying results in a way that is easy to share. It is based on the built-in version tracking capability in Ghidra and custom function matching algorithms, including some similarities with BinDiff [6].

QBinDiff is similar to the other diffing tools but aims to create a more modular framework that can be fitted to specific scenarios [41]. Graph-based structural matching is combined with graph node attributes, and used in a machine learning algorithm to calculate a mapping between the binaries' structure [41]. QBinDiff is more resource-demanding than BinDiff and is considered an experimental tool that requires more knowledge for optimal usage [41].

DeepBinDiff [42] is a prototype binary diffing framework using machine learning and an unsupervised neural network algorithm. Like QBinDiff, it leverages a structural matching of control flow graphs and semantic information from the basic blocks as the attributes for training the model [42].

Diaphora [43] is another open-source tool for binary differentiation using syntax and structural matching, and is considered the industry standard according to [45]. For syntax matching, Diaphora compares several hashes generated from bytes, instructions, and names, as well as mnemonics, assembly code, and constants [43]. Structural-based matching uses control flow graphs, similar to the previous approaches. Diaphora also has pseudo-code diffing and heuristics to leverage decompilation features in tools such as IDA Pro, where the assembly code is translated to a C-like pseudo-code [43]. Diaphora first finds all exact matches before finding partial matches and calculating similarity ratios [45]. The ease of automating the binary differentiation process and interacting with the results presents an advantage with Diaphora as it enables easier integration in binary analysis processes.

2.4 Malware behavior and capabilities

The Pyramid of Pain [46] emphasizes the effectiveness of responding to threat actors' tactics, techniques, and procedures (TTP). TTPs are more challenging to change than indicators such as hashes, IP addresses, and domains, but also more challenging to identify. Identifying behaviors and techniques in software could therefore be an effective way of identifying maliciousness and detecting malicious code in updates and legitimate software.

MITRE ATT&CK® is a widely used cyber threat modeling framework and knowledge base, originally intended as a structured way of emulating threat actors in exercises [47]. ATT&CK has other use cases as well, such as categorizing and labeling activity in cyber attacks and malicious behavior in systems through behavior analysis [47]. Behavior can be categorized as techniques used as part of a tactic to achieve an objective [48]. Using data sources in systems and networks to monitor behavior and categorizing them using ATT&CK, could aid in detecting malware and intrusions [47].

The Malware Behavior Catalog (MBC) [49] is based on MITRE ATT&CK, but is designed specifically for malware analysis. It is similarly structured, using objectives, behaviors, and methods, instead of tactics, techniques, and sub-techniques [49]. The MBC behaviors and objectives are more specified towards malware behavior, for example, the objective "anti-static analysis" includes a behavior "executable code obfuscation". One use case mentioned in [49], is similarity analysis, which is highly relevant for this thesis.

In 2020 the Mandiant FLARE team released the open-source malware analysis tool called *Capa* [14]. Capa identifies program capabilities through feature extraction and rule matching in PE, ELF, and .NET executables [8, 50, 51]. The features are derived from basic and advanced static analysis and include; strings, file header information, imported libraries, exported functions, section names, disassembly API calls, instruction mnemonics, and code references [8]. For .NET files, Capa extracts features such as; namespace, class, api, import, function-name, number, and string [51].

Capa uses a set of rules and signatures to associate program features with known techniques and behavior. Many of the rules and signatures are associated with the MITRE ATT&CK framework and the Malware Behavior Catalog (MBC) but may identify more capabilities due to the rules not being mapped. One Capa rule may also map to both ATT&CK and MBC. The Capa rules are defined by features and logical combinations of their values. The rule scope defines whether to match on basic block level, function level, or file level. [50]

Library functions included in the binary are matched using the same method as Hex-Rays' IDA Pro FLIRT signatures [50, 52]. The Capa rules can then match on library functions but the analysis will not be run on those matched functions [50].

Capa can be used as a standalone program to generate reports of identified capabilities and behaviors in analyzed executable files, or it can be used as a Python library as part of a workflow. Recently, the possibility of integrating a malware sandbox to get features from basic dynamic analysis has also been implemented [53].

2.5 Malware scoring

A malware score, also called severity or malice score, represents a value or scale that attempts to determine the threat of potential malware [54, 55]. It can aid analysts and defenders in threat assessment and incident response by providing a way of prioritizing actions and analysis resources [54]. Automated malware analysis tools, such as sandboxes which execute programs in a safe environment and report on the behavior, often provide a severity score based on the behaviors observed [54].

Existing research [54–56] addresses the limitations and flaws of existing scoring methods, and proposes improvements and new methods. They argue that existing methods mainly depend on the frequency of observed indicators and behaviors defined by signatures. These signatures define a severity score and, in some cases, a confidence score for the observed indicator or behavior, and are most often defined manually by researchers and domain experts [55]. Rohini et al. [54] presents an approach for scoring malware behavior by leveraging contextual information about behaviors occurring in relation to each other. Walker et al [56] emphasizes the potential of using threat intelligence sources to provide better confidence to the severity scores and signatures. They argue that indicators linked to previously reported attacks or threat actors could be leveraged to generate more robust and accurate malware scores.

MITRE Engenuity Center for Threat Informed Defense [57, 58] has published a framework for prioritizing ATT&CK techniques, intended as a systematic method for defenders to determine which techniques are most relevant to focus on. Essentially, it is a calculator where users define their network and systems, and which monitoring coverage is available to identify techniques [57]. The calculator also takes into account technique prevalence and choke points when calculating the weight for each technique [58]. Prevalence is based on the MITRE Engenuity Sightings Ecosystem [59], providing a database consisting of techniques observed over time based on reporting and community contributions [58]. Choke points are determined by examining technique relationships and finding the bottlenecks where many techniques lead to or techniques that are precursors for many others [58]. This project is not designed to provide a malware score but does present a weight for ATT&CK techniques indicating its prevalence and potential impact, taking into account some context of other related techniques [58]. Thus, it could present a possible improvement to existing malware scoring methods.

Chapter 3

Methodology

The goal of this thesis was to create a new approach for identifying malicious behavior in software updates from closed-source software supply chain attacks and determine if we could detect these attacks. The approach is based on existing tools and techniques to create an automated method of achieving our goal and thereby improving detection and reducing workload for analysts. Our method uses binary differentiation to find the new and updated code in software updates before we attempt to identify malicious behavior in this code. The identified behavior is further used to calculate a malicious score and train and test machine learning models for classification.

To perform the experiment, we created a dataset consisting of samples from disclosed closed-source software supply chain attacks and benign software samples. Because we are examining software updates, each element in the dataset consisted of two different versions of the same program, which we call a sampleset. Thus, for the malicious samplesets (closed-source software supply chain attacks), a benign version was grouped with the trojanized sample. The dataset creation is further described in detail in section 3.6.

This chapter presents the methodology and describes how the experiments were conducted and how the datasets were generated. First, an overview and description of the overall experiment is presented before we describe each stage in detail. Finally, we present the datasets used in the experiments and how they were created.

3.1 Experiment description

The experiment consisted of four stages and was performed on each sampleset in the dataset:

1. **Binary differentiation**: Perform automated binary differentiation to find the new and modified functions in each sampleset.
2. **Behavior identification**: Identify behaviors and capabilities from the functions that are not identical, for each sampleset.

3. **Malicious scoring**: Calculate a malicious score for each sampleset based on the extracted behaviors.
4. **Classification**: Use the identified behaviors as attributes in machine learning classification, to determine their usability in classification of malicious updates.

The stages are visualized in figure 3.1 below:

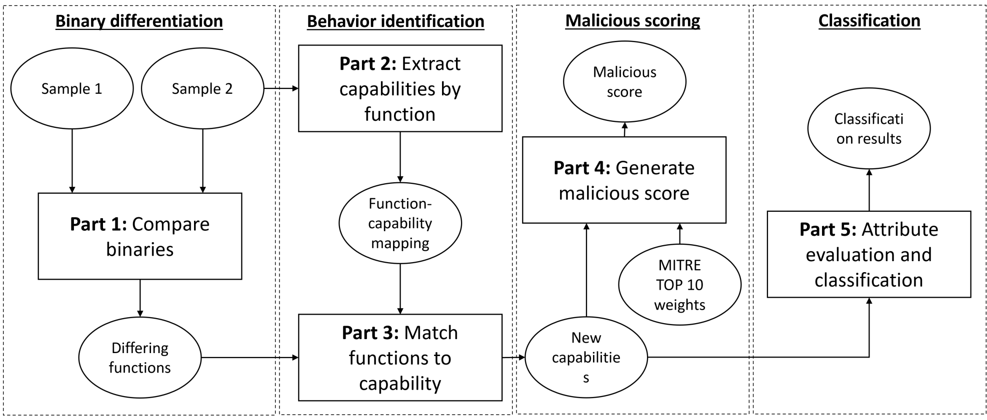

Figure 3.1: Method used in the thesis experiment

Stage 1 and 2 aims to answer the first research question, about how program capabilities can be identified from the changes introduced in program updates. Binary differentiation (stage 1) was used to find the changes in software updates. Taking two versions of the same program as input and finding the functions that differ. Behavior identification (stage 2) was completed in two parts. First, we found behaviors and capabilities for all functions in the second (newest) sample. Then, the functions that were equal were filtered out so we were left with only the functions that differed and the behavior found in them.

After performing the experiment on the whole dataset, the resulting data from stage 2 included all identified behaviors and capabilities for each sampleset. Thus, providing the results to discuss and answer the second research question, about the difference in prominent capabilities in benign and malicious updates. For the malicious samplesets, we examine the results from binary differentiation and behavior identification in more detail and discuss how they relate to existing reporting.

Stage 3 aimed to provide results for answering the fourth research question. The behaviors and capabilities identified for the new and modified functions were used to calculate a malicious score. The malicious score was calculated using weights for each capability that was mapped to a MITRE ATT&CK technique. This is covered in more detail in section 3.4.

The last stage used the identified behaviors as attributes in machine learning classification. Three attribute evaluation metrics were tested and three different classifiers were used to determine to which extent the behaviors and capabilities were able to classify malicious updates. The results from this stage would aid in answering the third research question.

3.2 Stage 1: Binary differentiation

To find the differences between the samples in each sampleset, we used binary differentiation. This provided the ability to find code differences and implementations to determine which functions are modified or novel in a program update.

The binary differentiation tool, Diaphora [43], was used to conduct the binary differentiation. It relies on Hex-Rays IDA Pro disassembler to generate the disassembly and function call graphs. We tested other disassemblers and binary diffing tools on the SolarWinds sample but found them lacking in the ability to disassemble .NET files, or more challenging to work with the result databases and integrate into the experiment workflow.

Diaphora creates an SQLite database for each of the samples with the information from the disassembler and then runs several matching strategies and heuristics using these databases. The output of this step was a new database (Diaphora file) consisting of one table listing the unmatched functions, i.e., new functions that are not present in the other binary. The other table of interest is the "results" table, containing the functions that are partial matches and their similarity ratio. The similarity ratio is a number between zero and one, where the value one indicates a complete match.

The database tables were queried to extract the new and modified functions. If the similarity ratio was less than one, indicating an unequal function, and it contained more than one single basic block, the function was labeled as modified. Functions with only one single basic block and a high similarity ratio yielded very minor differences, which were not of interest in this research.

To conduct the binary differentiation, we created the script presented in code listing A.2.

3.3 Stage 2: Behavior identification

Behavior identification was conducted using Mandiant's Capa library [14]. The Capa GitHub repository[14] provides a script for showing capabilities by function. By modifying this script, we were able to extract the capabilities for the last sample in each dataset and filtered out the functions that were not found to be new or modified by the previous binary differentiation stage. The modified script is provided in listing A.5. The results were written to the Diaphora database of each sampleset, creating a table for each behavior framework; ATT&CK techniques, MBC identifiers, and Capa capabilities. These tables were used to perform frequency analysis of the extracted capabilities to determine if they can provide more knowledge about SSCA. Examining the distribution of behaviors, techniques, and capabilities for the software updates, provided the foundation for answering the second research question. The ATT&CK table was additionally used to generate a malicious score, which we describe in the following section below.

3.4 Stage 3: Malicious scoring

We wanted to find out to what extent we could use identified behaviors to create a score indicating whether the software update was malicious or benign. To achieve this, we used the MITRE Engeneuity Top ATT&CK Techniques framework [58]. This experiment used the technique weights defined by prevalence score and a choke point score from the methodology that was provided in their public dataset [58]. The weights were used to score the significance of the extracted behaviors, making it possible to calculate a score indicating the maliciousness of the software update. To conduct the scoring, we used the spreadsheet provided by MITRE [59] without adjusting the weights for detection coverage. This provided weights for the techniques, ranging from 0 to 2.91. The complete table of weights are presented in appendix C.

For each sampleset, the techniques extracted in Stage 2 were summarized and multiplied by the weights from the spreadsheet. Finally, summarizing all the technique scores provided the final malicious score for the sampleset. The malicious score for each sampleset is defined as follows:

$$S = \sum_{i=1}^{N} n_t \cdot w_t$$

Where N is the number of identified techniques, n_t is the number of occurrences for a given identified technique t, and w_t is the weight value for that technique.

The malicious scoring results were also stored in the Diaphora database for each sampleset. One table for the score and another consisting of the identified techniques, their occurrence, and their weight. Thus, providing a way of examining how the score was generated. To answer research question 4, we examined the distribution of malicious scores across both datasets. Thus, determining whether the SSCA samplesets were significantly different from the benign dataset, based on the scores.

3.5 Stage 4: Classification

The experiment results are presented in digital databases, providing the following data for each sampleset: Number of functions, file size, MITRE ATT&CK techniques, MBC identifiers, Capa rules capability descriptions, scoring table, and the final malicious score.

To answer the research questions and contribute more knowledge to the domain of closed-source software supply chain attacks and malware analysis and detection, some statistical and machine learning measures were applied to the experiment results. A comparison of ATT&CK, MBC and the Capa capabilities was done to acquire knowledge of which framework and behaviors may be better suited for classifying updates as malicious or benign. The file size and the number of functions for each sampleset were also analyzed to determine the influence it could have on the amount of extracted capabilities or the malicious score.

3.5.1 Attribute evaluation

To determine the quality of the extracted capabilities and how they perform in classifying software updates as malicious or benign, we performed some attribute measures. Each extracted ATT&CK technique, MBC identifier, and capability represent an attribute. The class, or label, is represented by the samples' type, i.e., benign or malicious classification. To conduct these measures, the data analysis tools WEKA [60, 61] and RapidMiner[1] are used. The measures used are:

1. Correlation feature ranking using WEKA.
2. Attribute relevance using Chi-square, information gain, and correlation in RapidMiner.

Correlation feature ranking calculates Pearson's correlation between the attribute (technique) and the class (benign or malicious label). Chi-square statistic calculates a match between the observed frequencies of the attribute to a theoretical expected frequency. Information gain is a measure based on entropy to determine how much information each attribute holds. This is a standard measure and may weigh attributes with several unique values very high. Normalizing these weights would counter the bias but may then create a new bias to attributes with lower entropy [19].

3.5.2 Classification and validation

To evaluate whether the identified behaviors are suitable for classifying software updates as malicious or benign, we conducted three classifications using WEKA:

1. Naive Bayes
2. Multi-layered perceptron (MLP)
3. Random forest

All classifications were tested using 10-fold cross-validation due to the low number of samples. Cross-validation is often used when the dataset is small and dividing the dataset into a training and testing dataset is not suitable [19]. For 10-fold cross-validation, the dataset is divided into 10 subsets. A classification model is built for each subset and tested against the combined set of the other subsets [19]. The average from the 10 classification tests presents the final classification results.

Naive Bayes is a classifier using the principles of probability, but assuming the conditional independence of attributes [19]. In the learning stage, the overall probability of each class is calculated as the prior probability and the conditional class probabilities are calculated for each attribute conditional to the class [62].

[1] https://altair.com/altair-rapidminer

Thus, each attribute will have a calculated probability of being present for each class, making the Naive Bayes algorithm simple and fast [62].

Random forest is a type of decision tree algorithm where several decision trees are generated using the dataset and a random selection of attributes [19]. When testing the Random forest model, each decision tree gives a vote for the classification which together results in a final classification [19]. Random forest improves normal decision trees as the variance is decreased and mitigates the problem decision trees have with being too specific to the data used to train it [19].

Multi-layered perceptron is a feed-forward artificial neural network with multiple hidden layers of neurons [19]. In the learning phase, it starts with random weights for the layers but adjusts these weights through back-propagation for each learning iteration [19]. Thus, ending up with a model that has suitable weights for each layer to make as accurate predictions as possible.

The classifications were completed using each of the capability frameworks as attributes in WEKA. Three datasets were created, one for each of ATT&CK, MBC, and Capa capabilities. All three were populated with all samplesets, providing their type as a classification label, indicating if it was malicious or benign. The datasets were further populated with each of the capabilities and the frequency per sampleset. The names for the extracted capabilities represented the attributes in the dataset.

The Naive Bayes classifier was run with default settings in WEKA.
The random forest classification was run in WEKA with the following scheme:
weka.classifiers.trees.RandomForest -P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1
The MLP classification was run in WEKA with the following scheme:
weka.classifiers.functions.MultilayerPerceptron -L 0.1 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a. The number of hidden layers is set automatically based on the number of attributes and classes $layers = (attributes + classes)/2$.

3.6 Datasets

The dataset generation and considerations are presented in detail in this section. In this thesis, two datasets were created, combined, and used as input for the experiment. One malicious dataset, which consisted of samples from known closed-source software supply chain attacks, and one dataset with only benign software updates. Each element in the datasets consisted of two binaries from the same program, but different versions. These elements are called samplesets. The samplesets in the malicious dataset consisted of a benign version of the program that was trojanized in the supply chain attack and the trojanized version. The samplesets in the benign dataset consisted of two benign versions of the same program, but different versions. Figure 3.2 below illustrates the dataset composition.

Malicious dataset	**Benign dataset**
Sampleset 1 (benign version, trojanized version)	Sampleset 1 (benign version 1, benign version 2)
Sampleset 2 (benign version, trojanized version)	Sampleset 2 (benign version 1, benign version 2
: :	: :
Sampleset 10 (benign version, trojanized version)	Sampleset 420 (benign version 1, benign version 2

Figure 3.2: Dataset composition

3.6.1 Malicious SSCA dataset

The SolarWinds and M.E.Doc closed-source software supply chain attacks were part of the initial motivation for this thesis, due to their sophistication and impact. Thus, our malicious dataset was created with these and similar closed-source software supply chain attacks. M.E.Doc and SolarWinds are software written in C# .NET, however, we did not find other closed-source software supply chain attacks written in C# .NET. Therefore, the rest of the samples in the malicious dataset were C/C++ binaries.

The rest of the sample selection is based on the previous work of [11] and [12], and the publication from The Atlantic Council [63] which provides a description of 250 software supply chain attacks and disclosures. We aimed to find samples that were similar to SolarWinds and M.E.Doc using three criteria: The type of system that was targeted, the distribution vector, and type of software origin or type of codebase. The target system is the Windows operating system. The distribution vector is update hijacking. The software origin or codebase is third-party closed-source applications. The availability of samples was also a factor restricting the number of samplesets in the malicious dataset.

Filtering this dataset based on the criteria above and availability of samples, the dataset for this thesis consists of the software supply chain attack samples and a similar benign version shown in 3.1.

The SSCA samples are gathered from Recorded Future Triage sandbox [64] and malware database and VirusTotal [65] with the help from CrossPoint Labs. The main challenge in creating this database was finding suitable benign versions preceding the malicious updated binary. The attempt to find correct versions was aided by Intezer [66], which can present related samples based on automated analysis. Without an enterprise subscription, not all related samples where visible and very limited information was presented. However, we were able to leverage this to find relevant benign versions for some samplesets. Finding the immediately preceding version was not possible for all samplesets, and therefore some are either a few versions prior or behind the malicious.

The trojanized SmartPSS binary uses a slightly modified legitimate Windows

Table 3.1: The benign and malicious samples in the malicious SSCA dataset

Benign versions		
Software	**Version**	**MD5 Hash**
CCleaner	5.32.00.6129	68ddcb629a7f2c5a3d2392f8177a3cd0
M.E.Doc	1.0.0.0[1]	23fdc5d07b0a7d743137cce040345ba2
SolarWinds	2019.4.5200.9045	6b5f205d79a647b275500597975314a5
SwissRanger installer	1.0.14.706	6120d14f8bb27b469724333947d5717e
eGrabit installer	3.1.0.85[2]	8a6783a0b5cff2932b35b8c58925f5ab
eCatcher installer	4.3.0.15531[2]	877848de6f2135e2dbc7d036f6804528
SmartPSS installer	V2.002.0000009 .0.R.190426[2]	51ebe0db8fabace8ebc9d005b3c6cdec
SmartPSS mshta.exe	11.00.14393.2007	5ced5d5b469724d9992f5e8117ecefb5
3CX ffmpeg	18.11.1213	f459ce9af5091bc1e450eb753f6eb0b7
3CX d3decompiler	18.11.1213	cb9807f6cf55ad799e920b7e0f97df99
Malicious versions		
Software	**Version**	**MD5 Hash**
CCleaner	5.33.00.6162	ef694b89ad7addb9a16bb6f26f1efaf7
M.E.Doc	01.188-10.01.189	3efe62f6cb7285153114f888900a0962
SolarWinds	2019.4.5200.9083	b91ce2fa41029f6955bff20079468448
SwissRanger	1.0.14.706	e027d4395d9ac9cc980d6a91122d2d83
eGrabit	3.0.0.82	1080e27b83c37dfeaa0daaa619bdf478
eCatcher	4.0.0.13073	eb0dacdc8b346f44c8c370408bad4306
SmartPSS installer	V2.002.0000007 .0.R.181023	1430291f2db13c3d94181ada91681408
SmartPSS mshta.exe	11.00.14393.2007	c180f493ce2e609c92f4a66de9f02ed6
3CX Desktop App ffmpeg	18.12.416	74bc2d0b6680faa1a5a76b27e5479cbc
3CX Desktop App d3decompiler	18.12.416	82187ad3f0c6c225e2fba0c867280cc9

[1] Version defined in ZvitPublishedObjects.dll, unknown MeDoc version
[2] Benign version is a later version than the malicious

executable "mstha.exe" [33]. The benign, unmodified version was collected from The Windows Binaries Index [67].

Regarding the 3CX Desktop App, the malicious dataset includes the two trojanized DLLs modified in the installer, *ffmpeg.dll* and *d3decompiler_47.dll* [36]. From the Dragonfly campaign [32], the installers for eGrabit, eCatcher, and SwissRanger are included in the dataset. The malware inserted in these installers are not modified legitimate files, but rather standalone malware [32]. Thus, only the installers are included, which facilitate for the execution of the malware.

3.6.2 Benign dataset

The second dataset consists of a large number of Windows executable files developed with the .NET Framework [68]. The dataset is chosen based on its availability and the much faster processing of .Net binaries compared to other low-level programming languages such as C/C++. The main focus in the malicious dataset were the .NET files of SolarWinds and M.E.Doc, which also contributed to the choice of using .NET programs in the benign dataset.

This dataset [69] is derived from GitHub and labeled as benign. However, although it is labeled as benign, the repository owner does not provide explicit detail on sources or how they are classified as benign. Analyzing the dataset, it seems to be sourced from SourceForge, CNET, Microsoft and Softonic. Therefore, it is not guaranteed that the binaries are not malicious or contain adware. As a countermeasure the dataset was scanned with Microsoft Defender, yielding no malicious files. Furthermore, the VirusTotal verdicts for each file in the final dataset were collected, resulting in only four samplesets in the benign dataset being eliminated from the results. Thus, the remaining dataset consisted of samplesets detected by seven or less anti-virus engines on VirusTotal, most with zero or one detection. None were tagged as suspicious. See code listing A.4 for the script used to gather the VirusTotal information.

The benign dataset was created by finding binaries with the same program name but with different content and version description. This process was completed in the following steps:

1. Extract original program name, version and SHA256 hash from every binary.
2. Group together the files with the same program name and remove duplicates by hash.
3. Sort by version.
4. Create a list of tuples consisting of two consecutive files; one program version and the next in the list.

The script for creating the dataset is presented in listing A.1

Each of these tuples are referred to as a sampleset, with all samplesets making up the total dataset. Binary differentiation was conducted on each sampleset as part of the experiment, creating one similarity database per sampleset. This database consists of the binary differentiation results, made up of tables of unmatched (new) functions and matched functions. The matched functions table consists of a ratio from zero to one, indicating the similarity where one indicates an identical match. The dataset is further filtered by removing the samplesets were there are no matches, indicating that they were not versions of the same program. The filtering process is completed using the script in listing A.3. From the original dataset of 14397 samples, the final dataset consists of 420 samplesets after sampleset creation and filtering.

Using a dataset consisting of only .NET binaries has the downside of excluding a large number of existing software written in other programming languages. However, due to the large difference in processing time and hardware resource requirements, it provides a benefit in making it feasible to conduct this experiment on a fairly large dataset. A disadvantage of this dataset generation method is that creating samplesets based on program name and version description from the files' header data, does not guarantee that the versions are directly adjacent or even the same program. We attempted to mitigate this by filtering out the samplesets which had no or very few similar functions. However, the version deviation is not accounted for and some samplesets may be several versions apart. Controlling this would be a very time consuming task, and would not be feasible in the time scope of this thesis. However, the extracted capabilities and behaviors will still be representative for benign software updates, even though the amount may not be representative for adjacent software versions. Software updates are not always conducted for every version, and therefore, this method and dataset still reflects real world applications.

Chapter 4

Results

In this chapter we present the result from the experiments. The experiment stages of binary differentiation and behavior identification were closely linked and are presented combined in the first section of this chapter, where we describe the identified behaviors after binary differentiation. These results will provide the knowledge to answer our first two research questions; on how we can extract capabilities from software update changes and determine which are typical for benign and malicious updates. Next, we present results from malicious scoring and classification, which leverage the identified behaviors to determine whether software updates are malicious or benign. Finally, more detailed results are presented for the malicious dataset in section 4.4, including specific behaviors, malicious scores, and differentiation results. Thus, providing information for discussing how our approach performed and why it performed the way it did.

4.1 Behavior identification

We first found the new and modified functions in the software updates using binary differentiation. Then, we used Capa to identify behavior from these functions and categorize them into MITRE ATT&CK techniques, MBC, and Capa capabilities. In this section, we present the results from identifying behaviors from all samplesets, both malicious and benign. We examine the frequency and distribution of behaviors and capabilities and highlight the differences between malicious and benign updates. This lays the groundwork for applying malicious scoring and classification as well as answering our research questions.

This section is divided into three parts where we present the results from MITRE ATT&CK techniques, MBC, and Capa capabilities. This allows us to differentiate between them to see if one is better suited for finding malicious behavior in closed-source SSCAs.

4.1.1 MITRE ATT&CK techniques

The techniques extracted from the datasets are presented in figure 4.1. We see some techniques that seem very common in the benign set, such as T1012, T1083, T1620, and T1082. These technique identifiers represent the following techniques, respectively: Query registry, file and directory discovery, reflective code loading, and system information discovery. The most prominent technique is the file and directory discovery (T1083), which occurs 2157 times in the benign dataset and 18 times in the malicious. This technique is identified many times in the benign samplesets with a very high malicious score. Technique T1213 represents "data from information repositories", and has a very high occurrence for the small malicious dataset. There are two techniques which are not observed in the benign dataset, but is observed in the malicious. These are T1129 (shared modules) and T1125 (video capture).

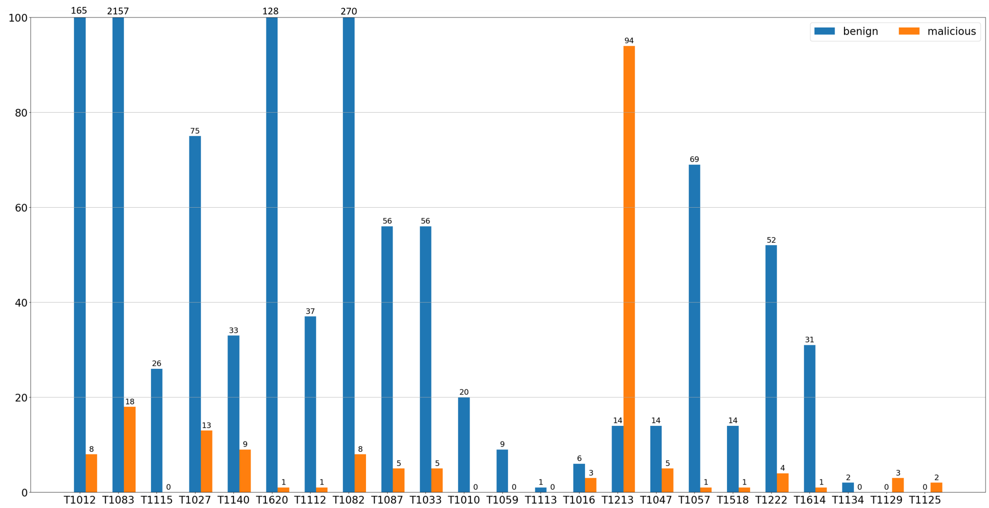

Figure 4.1: The histogram shows the distribution of MITRE ATT&CK techniques for both datasets

Figure 4.2 shows how the techniques are observed on average in each dataset. T1083 still stands out for the benign dataset, but T1213 shows the highest average occurrence of 9.4 times per sampleset. However, the results also show that T1213 is only present in two samplesets in the malicious dataset; the SolarWinds and M.E.Doc campaigns, with respectively, 25 and 69 occurrences. These are also the malicious samplesets with the highest score. For the benign dataset, the samplesets with the highest scores have a very large number of occurrences for the T1083 technique.

If we look at the highest weighted MITRE Top ATT&CK techniques (table 4.1) which we used for malicious scoring, we see that only T1112 and T1047 are observed in the malicious dataset.

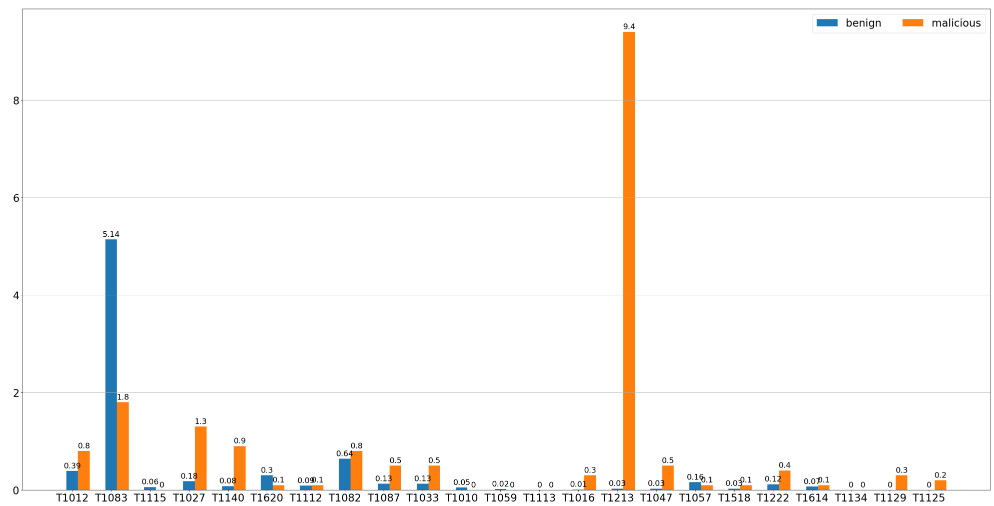

Figure 4.2: The histogram shows the average number of observed MITRE ATT&CK techniques for both datasets

Table 4.1: Top 10 ATT&CK techniques and their weights

Weight	ID	Description
2.914285714	T1059	Command and Scripting Interpreter
2.183333333	T1047	Windows Management Instrumentation
2.114285714	T1053	Scheduled Task/Job
1.945238095	T1055	Process Injection
1.880952381	T1218	Signed Binary Proxy Execution
1.826190476	T1574	Hijack Execution Flow
1.804761905	T1562	Impair Defenses
1.766666667	T1543	Create or Modify System Process
1.619047619	T1036	Masquerading
1.604761905	T1112	Modify Registry

4.1.2 Malware Behavior Catalog identifiers

When comparing the results from MBC behaviors to ATT&CK techniques across both datasets, MBC identifies more accounts of network communication, process interaction, and details in file system interaction. The use of WMI is not mapped to MBC, but is identified as an ATT&CK technique. The extracted MBC identifiers are presented in figure 4.3 by averaging the occurrences in benign and malicious datasets. The complete table of occurrences is provided in appendix B, table B.1.

Data encoding, cryptographic library usage, networking, and file attribution modification are behaviors averaging higher in the malicious dataset than in the benign. Like the ATT&CK techniques, file and directory discovery is more prominent for benign samplesets than for malicious ones. On average, process creation and termination also occur more frequently in benign than malicious.

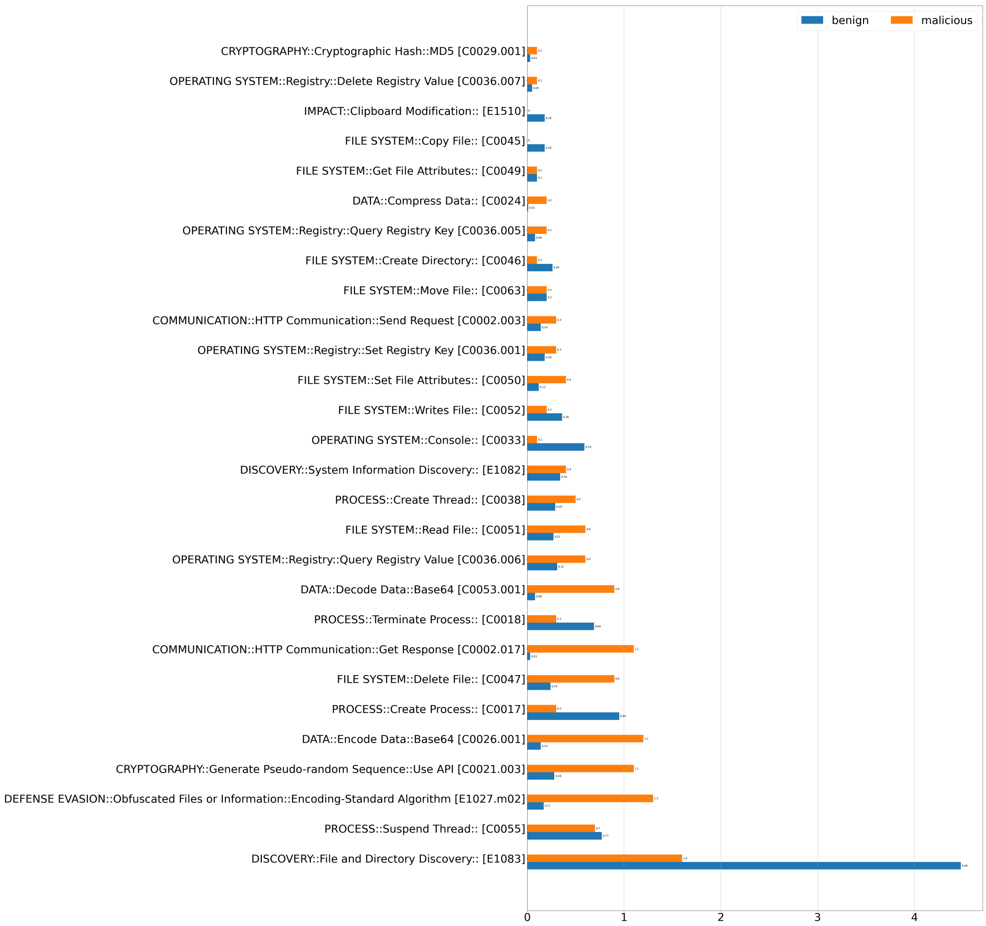

Figure 4.3: The histogram shows the average number of observed MBC identifiers for both datasets

4.1.3 CAPA capabilities

The Capa capabilities represent all extractions, including ATT&CK and MBC, as well as rules not mapped to these frameworks. Examining figure 4.4, the benign samplesets seem to have a higher average of capabilities identified with file system interaction, unmanaged runtime and memory, and process creation. The malicious samplesets have a higher average of data encoding, random number generation, WMI, network communication, and file attribution modifications. The complete table of Capa capabilities is provided in appendix B, table B.2.

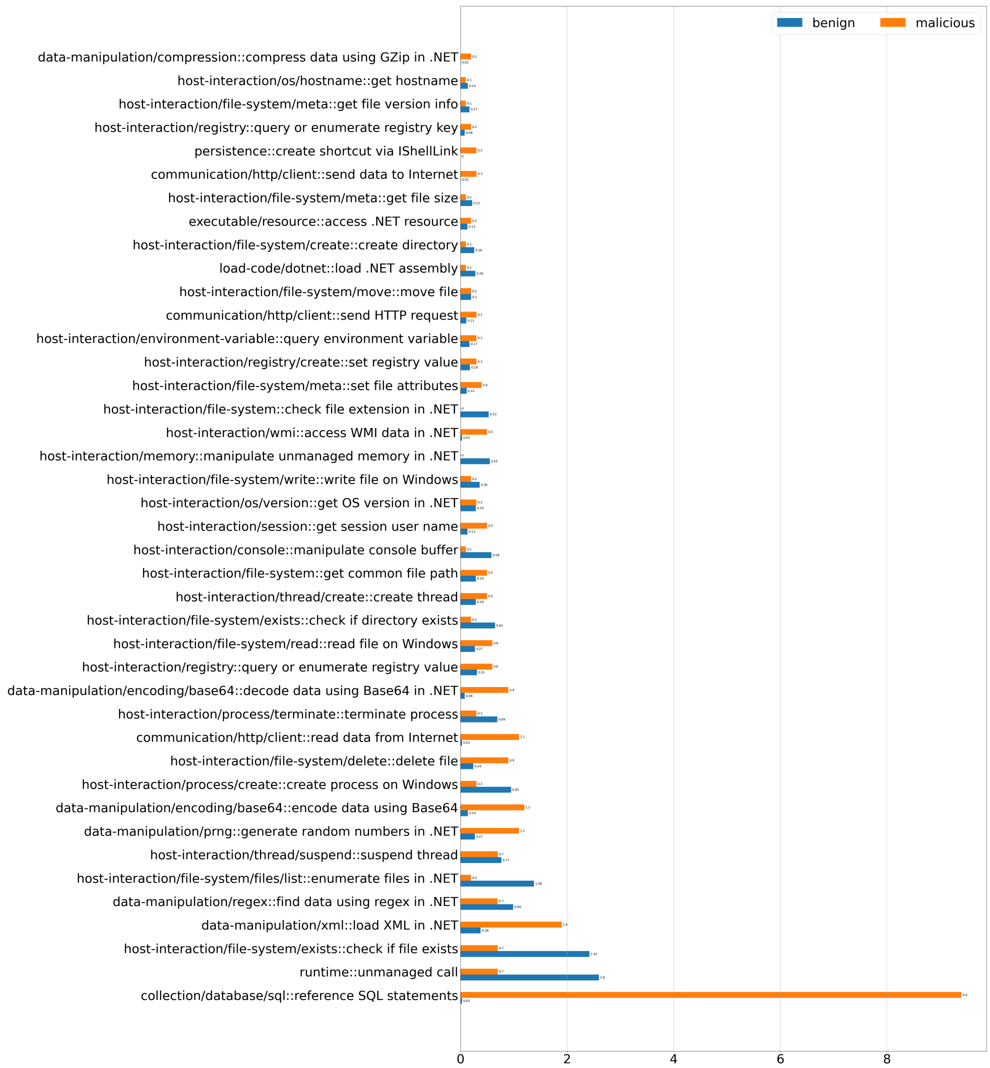

Figure 4.4: The histogram shows the average number of observed Capa capabilities for both datasets

4.2 Malicious score

The computed malicious score, based on the extracted ATT&CK techniques and the MITRE Top attack techniques weights, for the malicious SSCA samplesets are shown in table 4.2 below. The .NET binaries of M.E.Doc and SolarWinds provide relatively high scores compared to the other samples, which are written in C/C++.

Table 4.2: Malicious score for the SSCA campaigns

Program	Malicious score
Medoc	61.58
SolarWinds_Orion	35.70
CCleaner	2.82
SwissRanger_installer	0.00
eGrabit_installer	2.02
Talk2M_eCatcher_installer	2.02
3CXDesktopApp_d3decompiler	0.00
3CXDesktopApp_ffmpeg	1.49
SmartPSS_installer	1.15
SmartPSS_mshta	0.00

Figure 4.5 displays the scores for the 420 benign samplesets. The majority of the samples score below 3 points, with a total of 165 samples scoring 0. There are also a few outliers scoring very high. The basic statistical values for both datasets are shown in table 4.3 below. The benign scores average on 3.5 points, compared to the malicious dataset with 10.68. The standard deviation is much lower for the benign dataset, while it is relatively high for the malicious.

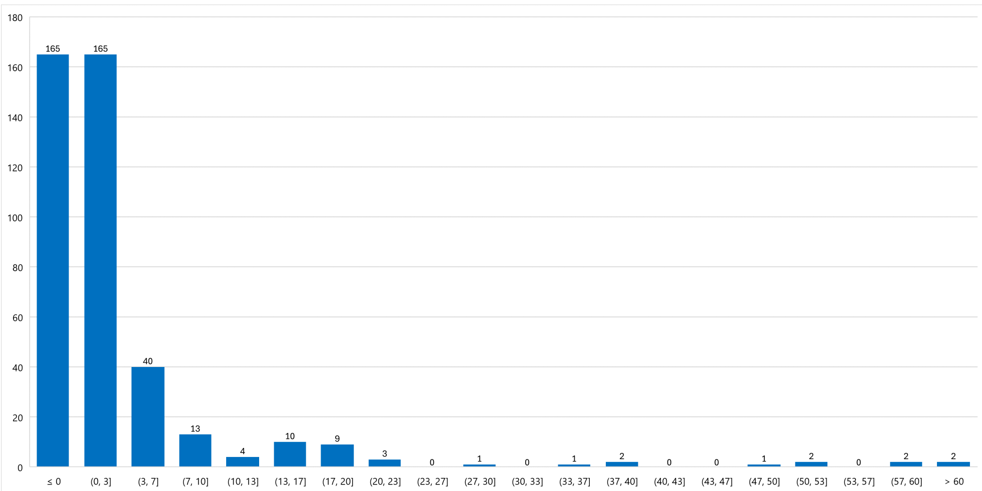

Figure 4.5: The histogram shows the distribution of the malicious score for the benign dataset. The Y-axis shows the number of samplesets and the X-axis shows malicious score ranges grouped in intervals of 3 points

Table 4.3 shows the malicious score statistics for both datasets.

Table 4.3: Malicious score statistics for both datasets

Dataset	Mean	Min	Max	Mode	Median	Std. Deviation	N
Benign	3.50	0	79.90	0	0.56	8.92	420
Malicious	10.68	0	61.58	0	1.76	20.94	10

Further analyzing the data, to see what influenced the malicious score, some correlations were found. We found that the number of functions in the binaries had a high correlation to the score. However, the *difference* in the number of functions between the versions in each sampleset did not seem to be correlated to the score. Neither did the size of the files. We also observed that the most frequently occurring techniques also correlated to the score.

4.3 Classification

4.3.1 Attribute evaluation

We ran different feature selection and evaluation metrics on the attributes of the results. These attributes included the malicious score, extracted techniques, behaviors, and capabilities. The dependent variable is the "type" labeling the samplesets as either malicious or benign. The metrics used were information gain, chi-square, and correlation.

The information gain evaluation indicates that none of the capabilities provide any significant value across all three capability frameworks (ATT&CK, MBC, and Capa capabilities). The largest value is 0.017 belonging to the capability of setting file attributes. The Chi-square evaluation results in values ranging from 0 to 84.4. The most significant attributes for classifying the samplesets are shown in table 4.4. Behavior involving obfuscation, encoding, cryptography, and network are scored highest. The most significant correlations between capabilities and the label (malicious or benign) are presented in table 4.5.

The high correlation features occur more frequently in the malicious dataset than in the benign, which we can see in the figures presented earlier. This could indicate that these features may be suitable for classification and determining whether updates are malicious or benign.

Table 4.4: Most significant capabilities based on chi-square values above 20.

Identifier	Value
T1213	84.39
collection/database/sql::reference SQL statements	84.39
data-manipulation/xml::load XML in .NET	42.55
T1027	42.42
DEFENSE EVASION::Obfuscation::Encoding-Std Algorithm [E1027.m02]	42.4
DATA::Encode Data::Base64 [C0026.001]	42.38
data-manipulation/encoding/base64::Base64 encode	42.38
data-manipulation/prng::generate random numbers in .NET	42.16
CRYPTOGRAPHY::Pseudo-random Sequence::Use API [C0021.003]	42.16
T1129	42.1
COMMUNICATION::HTTP Communication::Get Response [C0002.017]	42.1
DATA::Compress Data:: [C0024]	42.1
communication/http/client::read data from Internet	42.1
communication/http/client::send data to Internet	42.1
data-manipulation/compression::GZip compress in .NET	42.1
T1140	33.11
DATA::Decode Data::Base64 [C0053.001]	33.11
data-manipulation/encoding/base64::Base64 decode in .NET	33.11
FILE SYSTEM::Delete File:: [C0047]	20.47
host-interaction/file-system/delete::delete file	20.47
T1033	20.39
T1087	20.39
host-interaction/session::get session user name	20.39
communication/http/client::send HTTP request	20.20
T1047	20.16
host-interaction/wmi::access WMI data in .NET	20.16
T1016	20.11

Table 4.5: Most significant capabilities based on correlation values above 0.2.

Identifier	Value
DATA::Compress Data:: [C0024]	1
communication/http/client::send data to Internet	1
data-manipulation/compression::GZip compress in .NET	1
COMMUNICATION::HTTP Com::Get Response [C0002.017]	0.99
communication/http/client::read data from Internet	0.99
T1213	0.88
collection/database/sql::reference SQL statements	0.88
data-manipulation/xml::load XML in .NET	0.86
T1047	0.63
host-interaction/wmi::access WMI data in .NET	0.63
T1140	0.42
DATA::Decode Data::Base64 [C0053.001]	0.42
data-manipulation/encoding/base64::Base64 decode in .NET	0.42
communication/http::get system web proxy	0.33
DATA::Encode Data::Base64 [C0026.001]	0.32
data-manipulation/encoding/base64::Base64 encode	0.32
T1016	0.3
T1614	0.26
collection::get geographical location	0.26
COMMUNICATION::Socket Com::Create UDP Socket [C0001.010]	0.25
communication/socket/udp/send::create UDP socket	0.25
host-interaction/os/hostname::get hostname	0.24

4.3.2 Classification and validiation

To evaluate the potential for using the identified behaviors as attributes for the classification of malicious software updates, we used WEKA to run a Random Forest decision tree classifier, a multi-layered perceptron (MLP) neural network classifier, and a Naive Bayes classifier. 10-fold cross-validation was used and each classifier was executed for each of the behavior frameworks, where the identified behaviors were the attributes used.

In tables 4.6–4.8, the confusion matrices for the classification results are presented. Based on these values, precision and sensitivity is calculated and presented in table 4.9.

The results show that the malicious samplesets were more difficult to classify correctly with the highest sensitivity of 0.4 (4 of 10 samplesets). All classifications yielded relatively good results for benign samplesets, but this is not very useful if no malicious samplesets are correctly classified. MLP performs relatively well for MBC and Capa attribute sets, while Random Forest is not able to classify any malicious samples correctly. Naive Bayes performed close to MLP for the Capa attributes and better for the ATT&CK attributes. The best results were achieved with the MLP classifier and the Capa capabilities as attributes, where 4 malicious samples were correctly classified and only 5 benign samplesets were incorrectly classified. Thus, this model is only able to classify malware with a probability of 0.40, but the probability of wrongly predicting benign software updates as malicious is only 0.012.

The same classifications were run with the techniques and behaviors identified as the best features in the previous section table 4.4 and 4.5. These features yielded worse results, not being able to classify the malicious samples correctly. MLP was able to correctly classify one sampleset but had 1 incorrect benign. Naive Bayes incorrectly classified one benign and all malicious, while Random Forest incorrectly classified all malicious samples but all benign correct.

Table 4.6: Confusion matrices for the classification results using ATT&CK techniques as attributes.

Naive Bayes			**MLP**			**Random forest**		
Class	Benign	Mal	Class	Benign	Mal	Class	Benign	Mal
Benign	416	4	Benign	413	7	Benign	420	0
Mal	8	2	Mal	9	1	Mal	10	0

Table 4.7: Confusion matrices for the classification results using MBC behaviors as attributes.

Naive Bayes			MLP			Random forest		
Class	Benign	Mal	Class	Benign	Mal	Class	Benign	Mal
Benign	399	21	Benign	416	4	Benign	420	0
Mal	10	0	Mal	7	3	Mal	10	0

Table 4.8: Confusion matrices for the classification results using Capa capabilities as attributes.

Naive Bayes			MLP			Random forest		
Class	Benign	Mal	Class	Benign	Mal	Class	Benign	Mal
Benign	415	5	Benign	411	9	Benign	420	0
Mal	6	4	Mal	6	4	Mal	10	0

Table 4.9: Classification results for the different behavior frameworks presenting the precision and sensitivity for malicious and benign classifications

Classifier	**Class**	**ATT&CK**		**MBC**		**CAPA**	
		P	**S**	**P**	**S**	**P**	**S**
Random forest	**Benign**	0.977	1.0	0.977	1.0	0.977	1.0
	Malicious	0	0	0	0	0	0
Naive Bayes	**Benign**	0.981	0.990	0.976	0.950	0.986	0.981
	Malicious	0.333	0.200	0	0	0.333	0.400
MLP	**Benign**	0.979	0.983	0.983	0.990	0.986	0.988
	Malicious	0.125	0.100	0.429	0.300	0.444	0.400

4.4 Malicious software behavior

In this section, the results for the malicious dataset are presented in more detail. This allows us to evaluate and discuss the performance of our approach and highlight limitations and advantages. Table 4.10 below shows the occurrences of techniques in the malicious dataset.

Table 4.10: Malicious dataset technique distribution and malicious score

	Medoc	SolarWinds	Ccleaner	eGrabit	eCatcher	3CX	SmartPSS
T1012	2	4	0	1	1	0	0
T1083	8	4	1	2	2	0	1
T1115	0	0	0	0	0	0	0
T1027	10	1	1	0	0	1	0
T1140	6	3	0	0	0	0	0
T1620	0	1	0	0	0	0	0
T1112	0	1	0	0	0	0	0
T1082	4	4	0	0	0	0	0
T1087	4	1	0	0	0	0	0
T1033	4	1	0	0	0	0	0
T1010	0	0	0	0	0	0	0
T1059	0	0	0	0	0	0	0
T1113	0	0	0	0	0	0	0
T1016	2	1	0	0	0	0	0
T1213	69	25	0	0	0	0	0
T1047	0	5	0	0	0	0	0
T1057	0	1	0	0	0	0	0
T1518	0	1	0	0	0	0	0
T1222	0	0	1	1	1	0	1
T1614	0	1	0	0	0	0	0
T1134	0	0	0	0	0	0	0
T1129	0	0	2	0	0	0	1
T1125	0	0	0	1	1	0	0
Score	61.58	35.7	2.82	2.02	2.02	1.49	1.15

4.4.1 SolarWinds

In this thesis, the Sunburst backdoor from the SolarWinds supply chain attack has been analyzed. This is the malicious code injected into the build process of the Orion software dynamic link library (DLL) "SolarWinds.Orion.Core.BusinessLayer.dll" [28]. Based on the Microsoft analysis [28], our binary differentiation successfully extracted the added and modified malicious functions. However, other functions that were not mentioned in the analysis, were also identified as new or modified. The capability extraction shows that ATT&CK techniques were also extracted from the functions not explicitly identified as malicious. If we discard the techniques from the unreported functions, we can remove all T1213, one T1614, one T1082, and one T1083. This would result in a new score of 22.63. One of the highest weighted techniques, T1047, is still prominent with a frequency of 5.

The capability extraction also extracted MBC identifers, which show slightly differing behaviors compared to the ATT&CK techniques. The MBC identifiers for the SolarWinds sampleset finds more use of data encoding, cryptography, process and thread interaction, but misses WMI behavior.

Looking at the overall capabilities extracted, there are several not mapped to MITRE ATT&CK or MBC. However, comparing the capabilities to the techniques reported in [70], many of them are found in this research. Some are not identified, which is expected as the report includes techniques found from the context and not just from the malicious code. In addition, the Capa rules defining the techniques and the contributions to the report may also differ in interpretation by the authors. Causing the same behavior to be mapped to different techniques.

4.4.2 M.E.Doc - NotPetya

The supply chain attack on the M.E.Doc software is similar in the method of injecting malicious code into the source code of the file "ZvitPublishedObjects.dll". Based on the analysis by [71] and [5], it seems like the binary differentiation we did identified the malicious functions as new or modified. Like the SolarWinds sampleset, there are also additional functions identified as new or modified. These are not explicitly identified as malicious by the analysis. However, some of them are used by the malicious code, indicating that they may have been altered for malicious use. If we only include the program classes mentioned in the analysis reports, the M.E.Doc sampleset would get a score of 8.49, discarding most of the techniques found. However, as the malicious functions uses the other functions, this filtering would also cause a loss of information.

The extracted MBC identifiers are similar to the ATT&CK techniques but they are both missing some capabilities compared to each other, and to the overall capability extraction.

4.4.3 SmartPSS

The SmartPSS supply chain attack differs from SolarWinds and M.E.Doc, in the way the malicious code is introduced. In this campaign, the threat actors have created a new installer including a benign installer and a Windows executable. This executable is a benign Microsoft executable, mshta.exe, with a few bytes appended to it. The purpose of the file is to download a new file from a domain supplied on the command line by the installer script [33]. Thus, the sample does not contain modified code in the same way as the previous attacks but rather adds an executable file and a command line argument.

The differentiation step found some differences in functions for the installers and a few techniques were extracted, most likely not related to any malicious code, but rather the install software. Binary differentiation could not find any differences in the legitimate mshta sample compared to the slightly altered version. This is also expected knowing that only a few bytes were appended.

4.4.4 Dragonfly campaign

The SwissRanger, eGrabit, and Talk2M eCatcher supply chain attacks are attributed to the same campaign and consist of the same method for trojanizing the legitimate installers [32]. They are similar to the previous sample, providing a malicious binary with a legitimate installer. In this case, a malicious DLL is side-loaded to execute the malicious code.

The installers for eGrabit, and eCatcher provide the same results, while the installer for SwissRanger does not present any ATT&CK techniques but has two MBC identifiers. Running Capa to extract the capabilities from the known malicious DLLs, results in more behaviors identified and higher malicious scores. The eGrabit and eCatcher samples received a score of 9.18, while SwissRanger scored 19.68.

4.4.5 3CX Desktop App

The results for the 3CX Desktop App show that only one ATT&CK technique was identified in one of the trojanized DLLs. The technique is T1027, indicating obfuscated files or information. The MBC behaviors found in this sample also indicates obfuscation and base64 encoding.

These results were match what we described in the background, as the attack included two DLLs, one that loads the other obfuscated DLL. Thus, finding a few indicators of obfuscation in the first is expected. Our method was therefore unable to deobfuscate the other binary. However, this was not part of our scope and is left for future work.

Chapter 5

Discussion

In this chapter we discuss the results, what they mean, and how they compare to existing research. Throughout the discussion, we highlight factors that possibly impacted the results. Lastly, the research questions are answered based on the results and discussion.

5.1 Datasets

The very first part of the experiment was the dataset generation. The benign dataset was created with 420 samplesets, from a total of over 14000 .NET binary programs. It shows that generating a large dataset of software updates can be challenging, especially for closed-source software. Guaranteeing that every sampleset consists of two binaries of the exact same program and being versions directly following one another would not be feasible in the time frame of this thesis. One would have to compare each sample to the information from the distributors. However, the filtering after the binary differentiation guarantees a certain degree of similarity between the binaries in each sampleset, even though the versions may not be directly adjacent. Creating a dataset where the versions are known to be directly adjacent could be easier to accomplish by using open-source programs instead of closed-source, and may present a possibility of tracking software behavior across updates in a more precise manner. This was out of scope for this thesis and is left for future work.

Due to the low number of known closed-source software supply chain attacks, the malicious dataset included only two .NET binaries, the same type as the whole benign set. However, the eight other samplesets consisted of C/C++ samples from known closed-source SSCAs. These samples were either the program installer binary or a trojanized binary that we extracted from the installer.

For the samplesets where the attack included an added malicious binary that did not have a similar benign binary, we conducted the analysis on the entire installer. If the trojanized binary extracted from the installer had an equivalent benign version, those binaries were used in the samplesets.

We recognize that the behavior identification for the installers does not represent the behavior of the malicious program, but rather the installer and the changes done to the installer script. Thus, it is no surprise that we did not extract many capabilities from these samplesets. Using the malicious binaries inside the installers without having a benign version to differentiate, would not fit the methodology as binary differentiation would not be necessary. Therefore, we did not extract behaviors from those malicious binaries. An option would be to exclude these binaries, but due to the low number of malicious samples and the fact that they do alter the benign behavior they were included in the experiment.

The most significant challenge with this thesis was to find and use suitable samples from closed-source software supply chain attacks. Ideally, the malicious dataset would consist of only and more C# .NET binaries, but the prevalence of such amounts in this domain makes it infeasible. However, including the other samples presented results indicating that C/C++ software updates may provide fewer identified behaviors, which could be an interesting future research topic.

5.2 Binary differentiation

The binary differentiation was able to extract the functions with malicious code inserted by the threat actors. This shows that it could be a useful tool for reducing the amount of code to be analyzed, providing a more efficient way of looking for malicious updates and analyzing them. For the malicious dataset, we also identified new or modified functions that were not reported as malicious by existing analysis. The reason for this could be that legitimate updates were pushed alongside the malicious code. Particularly in the SolarWinds sampleset, where the versions are very close, the number of benign new functions was low. For the M.E.Doc sampleset, however, the version of the benign sample was not identifiable and may therefore include more benign updates, causing more functions to be identified as new or modified.

For the benign dataset, the sample size of 420 did not allow for the same detailed analysis. We would have to manually go through each sampleset to examine the function names and identified behaviors and find reliable information about the changes between each version to compare our results to. This would be a very time-consuming task and was not feasible in the time frame of this thesis. Even though we were not able to validate that the differing functions were correctly identified, our results showed that binary differentiation was able to find new and modified functions between updates.

Some of the samplesets with high scores and a large number of capabilities were further examined to see if this was due to a large difference in file size or the number of functions. However, they did not show a clear relationship to the file size difference or the difference in number of functions.

5.3 Capability and behavior identification

As mentioned in the article [50], the Capa rules are mostly mapped to the MITRE ATT&CK framework or Malware Behavior Catalog. Only using the ATT&CK results does not provide completely accurate identification of the behavior of the malicious code. The MBC behaviors presented more information about process and thread interaction, networking, use of cryptographic functions, data encoding, and file system interaction.

The objectives and behaviors in MBC are more tailored to executable programs and malware, which could explain the difference in extracted capabilities between ATT&CK and MBC. It will also depend on how the Capa rules are written, where the rule author must define the mapping to MBC or ATT&CK.

When examining the difference in the average frequency of behaviors for the malicious dataset and the benign dataset, a few behaviors are more prominent in the malicious dataset. Base64 encoding and decoding occur more often in malicious updates, which match the findings from [11] and [12] where obfuscation is a feature found in the trojanized updates.

Networking interaction, including reading and sending data over HTTP, is averaging higher in malicious updates. This behavior is not necessarily malicious but can give an indication on how potential command-and-control is implemented. Barr-Smith et al. [12] do not report on changes in network activity, but our approach is able to find specific types and protocols. Similarly, Refsnes [11] reports on identified IP addresses and domains, but not how they are communicated to, which we are able to. Thus, presenting an advantage of our approach compared to previous work, where we identify specific network protocols and details on how the network communication is conducted. Our approach was not able to identify specific IP addresses and domains and was not part of the objectives for this project. However, researching the possible advantages of combining these elements is left for future work.

5.4 Malicious scoring

For the malicious scoring, we expected to find higher scores for malicious updates than benign updates. The scoring based on the ATT&CK techniques did not provide a significant difference between the benign dataset and the malicious dataset. For the malicious updates only the .NET samplesets, SolarWinds and M.E.Doc, provided a high score. There were still some benign samplesets that scored similarly to these and even a few with a higher score. If we filtered out the outliers from the benign dataset, it could be possible to identify these two malicious updates from the score. These benign outliers with very high scores also seem to have a very high count of one or two techniques which impact the score significantly. To counter such challenges, a future modification to the model could be to account for high-frequency techniques.

The malicious samplesets that were not .NET presented relatively low malicious scores and we would not be able to identify them among benign .NET updates. This low score is due to a low number of identified techniques in the new or modified functions. Further research into software updates for programs written in lower-level languages could help explain this outcome.

The approach presented in this thesis shows that we can identify possible malicious .NET software updates, but may also include some benign outliers or false positives. Our approach saves the mapping between functions and extracted capabilities as well as how the malicious score is calculated. Thus, presenting an advantage where we can examine the resulting database to identify which functions perform the identified behaviors and how the score was calculated. This can provide the malware analysts with a good starting point for analysis.

The malicious scoring also shows the complexity in malware classification. Benign software can show behavior that we associate with malware and the other way around. One behavior is not necessarily malicious in itself and therefore we have to take into account the surrounding behaviors; the preceding and following actions. The malicious scoring uses technique weights that are calculated based on both prevalence and surrounding behavior [58]. To our knowledge, our research is the first to test the use of these weights in malware scoring. The results indicate that it is challenging to correctly calculate these weights, and that further research is needed.

5.5 Attribute evaluation and classification

The attribute evaluation conducted using the WEKA and RapidMiner software provided us with some information about how useful the different attributes were for classifying the samplesets. However, when using the top-rated Chi-square and correlation attributes, the classification results were worse than not using this selection.

For all three classifiers tested (MLP, Naive Bayes, and Random forest), the 10-fold cross-validation showed that at best we were able to classify 4 of 10 malicious software updates. For benign software updates, all classifiers performed well with precision and sensitivity of at least 0.95. The low number of malicious samples hampers the usability of such a model to classify malicious updates effectively. Ideally, we would have more malicious samples to build a better machine learning model and to be able to apply more tests. However, with the small dataset, we were still able to create a model that classified on average malicious samples with a probability of 0.4 and a probability of 0.012 of benign samples being incorrectly classified.

Even if the generated models using the current dataset is not applicable as the single system for malware classification, it could provide an additional depth to malware detection technology and reduce the workload for the malware analyst.

5.6 Research questions

5.6.1 RQ1. How can program behavior be extracted from the changes in program updates?

Our research shows that binary differentiation successfully identifies new and modified functions in software updates. Furthermore, we were able to identify program behavior from these functions using the Mandiant FLARE Capa framework. Using this method is effective on software updates, regardless of being malicious or benign. Our approach is also able to accomplish this regardless of the version difference between the samples.

Leveraging the Capa framework, we can extract capabilities and behaviors, and the majority of them are mapped to MITRE ATT&CK and MBC. The extracted capabilities from the malicious samplesets do not deviate far from reports when examining the SolarWinds and M.E.Doc attacks. However, if the differentiation identifies benign functions, capabilities from these functions are also identified.

5.6.2 RQ2. Which behaviors are prominent in benign software updates and how do they differ from malicious updates?

In this thesis, we created a benign dataset consisting of 420 .NET software updates using automated methods. This dataset was used with our approach to identify behaviors for each benign software update. Thus, providing results on the frequency and distribution of program behaviors for benign software updates.

From the experiment results, we found that the benign software updates have a wide range of behaviors. The most prominent based on the average occurrence, include unmanaged code calling, file enumeration, creating and terminating processes, and finding data using regex.

According to our results, the malicious updates have a higher average occurrence of the following behaviors: XML loading, random number generation, base64 encoding and decoding, file deletion, HTTP sending and receiving data, WMI usage, and file attribution modifications.

5.6.3 RQ3. To what extent can extracted behaviors be used to identify software supply chain attacks?

As presented in the results and discussed previously, this approach of extracting the behaviors and capabilities from software updates can be useful for detecting SSCAs. First, our approach identifies new and modified functions and then maps the behavior found in functions to standardized formats, including MITRE ATT&CK and MBC. This presents an advantage for reporting purposes and extensibility to other existing frameworks. Another advantage includes being able to easily examine behaviors for initial analysis and then have a starting point for potential further analysis, as we know the functions conducting the behavior.

For automatic detection of SSCAs, the tested classifiers show that we are able to detect malicious software updates with a probability of 0.4 while the probability of incorrectly classifying benign updates as malicious is as low as 0.012. Thus, our approach can present a potential addition to existing detection methods, with the advantage of providing useful knowledge for deeper analysis and reducing the workload for analysts.

5.6.4 RQ4. How can identified behaviors provide a malicious score that can be used to detect malicious software updates?

This thesis used existing provenance weights of MITRE ATT&CK techniques from the MITRE Top 10 Techniques [58]. We calculated a malicious score for each sampleset using these weights multiplied with the occurrence of ATT&CK techniques. For the benign dataset, most samplesets got a score between 0 and 3, with 0 being the most frequent. 90 of 420 samplesets scored above 3, with the highest score of 79.9.

The malicious dataset only consisted of 10 samplesets and had a very high standard deviation. The two .NET samplesets (SolarWinds and M.E.Doc) had high scores, but the eight C/C++ samplesets had relatively low scores between 0 and 3.

Due to the low number of samplesets in the malicious dataset, we cannot conclude that the malicious score is a reliable marker for all malicious updates. However, the results indicate that malicious .NET software updates could present scores that have a significantly higher value than the average benign updates. Therefore, indicating that the malicious scoring could be useful for detecting closed-source SSCAs for .NET software.

Chapter 6

Conclusion

In this thesis, we propose a novel automated approach for identifying malicious behavior in software updates and use these behaviors to generate a malicious score. The validation of this approach, using benign and malicious software updates, presents new knowledge of behaviors in closed-source software supply chain attacks and techniques that can be used to detect them.

The results show that our approach successfully identifies behaviors from new and modified functions in software updates, by leveraging existing tools and frameworks. The behaviors are reported in standardized MITRE ATT&CK techniques and Malware Behavior Catalog (MBC) identifiers and mapped to the functions conducting the behavior. This can aid in standardized reporting formats and provide the analysts with an advantage in triaging and a better starting point for advanced analysis.

Malicious scores were successfully generated from the identified ATT&CK techniques using existing weights from the MITRE Top 10 techniques [58]. The malicious .NET updates would be possible to distinguish from the benign .NET updates, as most benign updates scored below 3. However, malicious updates written in lower-level languages scored within the normal range of benign .NET updates. Software updates with a high frequency of one or few techniques receive a high score, even if the techniques are not necessarily malicious or suspicious by themselves. More research is, therefore, necessary to identify proper attribute weights and handling of high frequency but few techniques.

In this master thesis, we have successfully identified program behavior and capabilities in benign and malicious software updates. The behaviors more prominent in malicious updates are presented, showing a higher frequency of data encoding and obfuscation, random number generation, compression, network activity, and file deletion. The Malware Behavior Catalog (MBC) is found to be a more accurate framework in identifying behavior for software than ATT&CK. However, the use of both MBC and ATT&CK presents an advantage through the automated behavior extraction which can be used to report behaviors in a standardized format.

To further evaluate the use of identified behaviors to detect malicious software updates, we ran three machine learning classifiers with 10-fold cross-validation and the behaviors as attributes. We were able to generate a model that classified malicious software updates with a probability of 0.4 while retaining a probability of incorrectly classifying benign updates as malicious at 0.012. This model is not suitable as a single detection method but presents a possible addition to provide another layer of defence.

6.1 Future work

This thesis presents an approach to identify behavior in software updates and detect possible malicious updates as part of closed-source software supply chain attacks. As the approach leverages existing open-source tools and frameworks in a workflow, future work could include extending the methods and including other behavior or malware identification techniques.

Validating this approach using open-source software could also contribute to increased knowledge of software supply chain attacks and how we can detect them. Additionally, creating a similar benign dataset consisting of lower-level written programs, such as C/C++, would be an interesting topic to examine the differences in behavior identification across languages and runtime environments.

Using the attribute weights from the MITRE Top 10 techniques to create a malicious score, did not provide distinct results for effectively classifying malicious software updates. Thus, further research into the weighting of techniques specific to software supply chain attacks and examining their interdependence could provide further insight into the behavior of SSCAs.

Bibliography

[1] ENISA, "Enisa threat landscape 2023," ENISA, Tech. Rep., 2023. [Online]. Available: `https://www.enisa.europa.eu/publications/enisa-threat-landscape-2023` (visited on 05/20/2024).

[2] T. Herr, W. Loomis, S. Scott, and J. Lee, "Breaking trust: Shades of crisis across an insecure software supply chain," Atlantic Council, Tech. Rep., 2020. [Online]. Available: `https://www.atlanticcouncil.org/in-depth-research-reports/report/breaking-trust-shades-of-crisis-across-an-insecure-software-supply-chain/`.

[3] FireEye, "Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor," Mandiant, Tech. Rep., 2022. [Online]. Available: `https://cloud.google.com/blog/topics/threat-intelligence/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor` (visited on 05/18/2024).

[4] E. Brumaghin, W. Mercer, and C. Williams, "Ccleanup: A vast number of machines at risk," Cisco Talos, Tech. Rep., 2017. [Online]. Available: `https://blog.talosintelligence.com/avast-distributes-malware/` (visited on 05/22/2024).

[5] D. Maynor, M. Olney, and Y. Younan, "The medoc connection," CISCO Talos Intelligence Group, Tech. Rep., 2017. [Online]. Available: `https://blog.talosintelligence.com/the-medoc-connection/` (visited on 05/20/2024).

[6] J. McIntosh, "Ghidriff: Ghidra binary diffing engine," Tech. Rep., 2023. [Online]. Available: `https://clearbluejar.github.io/posts/ghidriff-ghidra-binary-diffing-engine/` (visited on 05/23/2024).

[7] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Comput. Surv.*, vol. 54, no. 3, 2021, ISSN: 0360-0300. DOI: `10.1145/3446371`. [Online]. Available: `https://doi.org/10.1145/3446371`.

[8] W. Ballenthin and M. Raabe. "Capa: Automatically identify malware capabilities." (2020), [Online]. Available: `https://www.mandiant.com/resources/blog/capa-automatically-identify-malware-capabilities` (visited on 05/16/2024).

[9] M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabber’s knife collection: A review of open source software supply chain attacks,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds., Cham: Springer International Publishing, 2020, pp. 23–43, ISBN: 978-3-030-52683-2.

[10] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, “Towards measuring supply chain attacks on package managers for interpreted languages,” *arXiv preprint arXiv:2002.01139*, 2020.

[11] M. W. Refsnes, “Exploring trojanized closed-source software supply chain attacks through differential malware analysis,” M.S. thesis, Norwegian University of Science and Technology (NTNU), 2023.

[12] F. Barr-Smith, T. Blazytko, R. Baker, and I. Martinovic, “Exorcist: Automated differential analysis to detect compromises in closed-source software supply chains,” in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED’22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 51–61, ISBN: 9781450398855. DOI: `10.1145/3560835.3564550`. [Online]. Available: `https://doi.org/10.1145/3560835.3564550`.

[13] A. Andreoli, A. Lounis, M. Debbabi, and A. Hanna, “On the prevalence of software supply chain attacks: Empirical study and investigative framework,” *Forensic Science International: Digital Investigation*, vol. 44, p. 301 508, 2023, Selected papers of the Tenth Annual DFRWS EU Conference, ISSN: 2666-2817. DOI: `https://doi.org/10.1016/j.fsidi.2023.301508`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S2666281723000094`.

[14] The FLARE Team, *capa, a tool to identify capabilities in programs and sandbox traces.* Jul. 2020. [Online]. Available: `https://github.com/mandiant/capa` (visited on 05/18/2024).

[15] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[16] L. Zeltser. “3 phases of malware analysis: Behavioral, code, and memory forensics.” (2010), [Online]. Available: `https://www.sans.org/blog/3-phases-of-malware-analysis-behavioral-code-and-memory-forensics/` (visited on 05/01/2024).

[17] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, “Deep learning for classification of malware system call sequences,” in *AI 2016: Advances in Artificial Intelligence*, B. H. Kang and Q. Bai, Eds., Cham: Springer International Publishing, 2016, pp. 137–149, ISBN: 978-3-319-50127-7.

[18] SentinelOne. “What is malware detection? | a comprehensive guide.” (), [Online]. Available: `https://www.sentinelone.com/cybersecurity-101/what-is-malware-detection/` (visited on 06/01/2024).

[19] I. Kononenko and M. Kukar, *Machine learning and data mining*. Horwood Publishing, 2007, pp. 153–167.

[20] Sonatype. "What are software dependencies?" (), [Online]. Available: https://www.sonatype.com/resources/articles/what-are-software-dependencies (visited on 05/31/2024).

[21] US National counterintelligence and security center, "Software supply chain attacks," 2021. [Online]. Available: https://www.dni.gov/files/NCSC/documents/supplychain/Software_Supply_Chain_Attacks.pdf (visited on 05/11/2024).

[22] IBM. "What is log4shell." (), [Online]. Available: https://www.ibm.com/topics/log4shell (visited on 05/31/2024).

[23] L. Grustniy. "Log4shell a year on." (2022), [Online]. Available: https://www.kaspersky.com/blog/log4shell-still-active-2022/46545/ (visited on 05/31/2024).

[24] W. Whitmore, "Today's attack trends — unit 42 incident response report," Palo Alto Networks Unit 42, Tech. Rep., 2024. [Online]. Available: https://www.paloaltonetworks.com/blog/2024/02/unit-42-incident-response-report (visited on 05/09/2024).

[25] C. I. S. Agency, "Defending against software supply chain attacks," 2021. [Online]. Available: https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508.pdf (visited on 05/20/2024).

[26] A. Greenberg and M. Burgess. "The mystery of 'jia tan', the xz backdoor mastermind." (2024), [Online]. Available: https://www.wired.com/story/jia-tan-xz-backdoor/ (visited on 05/01/2024).

[27] X. Wang, "On the feasibility of detecting software supply chain attacks," in *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*, 2021, pp. 458–463. DOI: 10.1109/MILCOM52596.2021.9652901.

[28] M. T. Intelligence, "Analyzing solorigate, the compromised dll file that started a sophisticated cyberattack, and how microsoft defender helps protect customers," Microsoft, Tech. Rep., 2020. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2020/12/18/analyzing-solorigate-the-compromised-dll-file-that-started-a-sophisticated-cyberattack-and-how-microsoft-defender-helps-protect/ (visited on 05/18/2024).

[29] CISA, "Petya ransomware," Tech. Rep., 2017. [Online]. Available: https://www.cisa.gov/news-events/alerts/2017/07/01/petya-ransomware (visited on 05/21/2024).

[30] L. Mathews. "Notpetya ransomware attack cost shipping giant maersk over $200 million." (2017), [Online]. Available: https://www.forbes.com/sites/leemathews/2017/08/16/notpetya-ransomware-attack-cost-shipping-giant-maersk-over-200-million/ (visited on 05/23/2024).

[31] A. L. Johnson, "Dragonfly: Western energy companies under sabotage threat," Broadcom, Tech. Rep., 2014. [Online]. Available: https://community.broadcom.com/symantecenterprise/viewdocument/dragonfly-western-energy-companies?CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68 (visited on 05/21/2024).

[32] J. T. Langill, "Defending against the dragonfly cyber security attacks," Belden, Tech. Rep., 2014. [Online]. Available: https://www.belden.com/hubfs/resources/knowledge/white-papers/Belden-White-Paper-Dragonfly-Cyber-Security-Attacks-AB_Original_68751.pdf?hsLang=en.

[33] T. McLellan, R. Dean, J. Moore, N. Harbour, M. Hunhoff, J. Wilson, and J. Nuce, "Smoking out a darkside affiliate's supply chain software compromise," Mandiant, Tech. Rep., 2021. [Online]. Available: https://www.mandiant.com/resources/blog/darkside-affiliate-supply-chain-software-compromise (visited on 05/20/2024).

[34] "3cx business communication solutions & software." (), [Online]. Available: https://www.3cx.com/ (visited on 05/26/2024).

[35] R. Falcone and J. Grunzweig, "Threat brief: 3cxdesktopapp supply chain attack," Palo Alto Networks Unit 42, Tech. Rep., 2023. [Online]. Available: https://unit42.paloaltonetworks.com/3cxdesktopapp-supply-chain-attack/ (visited on 05/21/2024).

[36] J. Johnson, F. Plan, A. Sanches, R. Fontana, J. Nicastro, D. Andonov, M. Fodoreanu, and D. Scott, "3cx software supply chain compromise initiated by a prior software supply chain compromise; suspected north korean actor responsible," Mandiant, Tech. Rep., 2023. [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/3cx-software-supply-chain-compromise (visited on 05/22/2024).

[37] T. M. Research, "Preventing and detecting attacks involving 3cx desktop app," Trend Micro, Tech. Rep., 2023. [Online]. Available: https://www.trendmicro.com/en_us/research/23/c/information-on-attacks-involving-3cx-desktop-app.html (visited on 05/22/2024).

[38] H. Flake, "Structural comparison of executable objects," in *International Conference on Detection of intrusions and malware, and vulnerability assessment*, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:15671919.

[39] M. Downie, "Decompilation of c# code made easy with visual studio," 2021. [Online]. Available: https://devblogs.microsoft.com/visualstudio/decompilation-of-c-code-made-easy-with-visual-studio/ (visited on 05/16/2024).

[40] Google, *Bindiff*. [Online]. Available: https://github.com/google/bindiff (visited on 05/22/2024).

[41] R. Cohen, R. David, and R. Mori, *Qbindiff: A modular diffing toolkit*, 2023. [Online]. Available: https://blog.quarkslab.com/qbindiff-a-modular-diffing-toolkit.html (visited on 05/22/2024).

[42] Y. Duan, X. Li, J. Wang, H. Yin, *et al.*, "Deepbindiff: Learning program-wide code representations for binary diffing," 2020.

[43] J. Koret, *Diaphora, the most advanced free and open source program diffing tool.* 2015. [Online]. Available: https://github.com/joxeankoret/diaphora (visited on 04/03/2024).

[44] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *Sstic*, vol. 5, no. 1, p. 3, 2005.

[45] S. Ullah and H. Oh, "Bindiff nn : Learning distributed representation of assembly for robust binary diffing against semantic differences," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, Jul. 2021. DOI: 10.1109/TSE.2021.3093926.

[46] D. Bianco. "The pyramid of pain." (2013), [Online]. Available: https://detect-respond.blogspot.com/2013/03/the-pyramid-of-pain.html (visited on 05/08/2024).

[47] B. Strom, A. Applebaum, D. Miller, K. Nickels, A. Pennington, and C. Thomas, "Mitre att&ck: Design and philosophy," *MITRE*, 2020. [Online]. Available: https://attack.mitre.org/docs/ATTACK_Design_and_Philosophy_March_2020.pdf (visited on 05/01/2024).

[48] A. E. Torres, F. Torres, and A. T. Budgud, "Cyber threat intelligence methodologies: Hunting cyber threats with threat intelligence platforms and deception techniques," in *2nd EAI International Conference on Smart Technology*, F. Torres-Guerrero, L. Neira-Tovar, and J. Bacca-Acosta, Eds., Cham: Springer International Publishing, 2023, pp. 15–37, ISBN: 978-3-031-07670-1.

[49] MITRE. "Malware Behavior Catalog." (), [Online]. Available: https://github.com/MBCProject/mbc-markdown (visited on 05/20/2024).

[50] W. Ballenthin, M. Raabe, M. Hunhoff, and A. M. M. Gomez. "Capa 2.0: Better, stronger, faster." (2021), [Online]. Available: https://www.mandiant.com/resources/blog/capa-2-better-stronger-faster (visited on 05/16/2024).

[51] W. Ballenthin, M. Raabe, M. Hunhoff, and A. Virgaonkar. "Capa v4: Casting a wider .net." (2022), [Online]. Available: https://www.mandiant.com/resources/blog/capa-v4-casting-wider-net (visited on 05/16/2024).

[52] Hex-Rays. "Ida f.l.i.r.t technology: In-depth." (2011), [Online]. Available: https://hex-rays.com/products/ida/tech/flirt/in_depth/ (visited on 04/19/2024).

[53] Y. Elhamer, W. Ballenthin, M. Raabe, and M. Hunhoff. "Dynamic capa: Exploring executable run-time behavior with the cape sandbox." (2024), [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/dynamic-capa-executable-behavior-cape-sandbox/ (visited on 05/30/2024).

[54] R. S, G. Ramesh, and A. R. Nair, "Magic: Malware behaviour analysis and impact quantification through signature co-occurrence and regression," *Computers & Security*, vol. 139, p. 103 735, 2024, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2024.103735. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824000361.

[55] M. S. Abbasi, H. Al–Sahaf, and I. Welch, "Automated behavior-based malice scoring of ransomware using genetic programming," in *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2021, pp. 01–08. DOI: 10.1109/SSCI50451.2021.9660009.

[56] A. Walker, M. F. Amjad, and S. Sengupta, "Cuckoo's malware threat scoring and classification: Friend or foe?" In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, 2019, pp. 0678–0684. DOI: 10.1109/CCWC.2019.8666454.

[57] C. for Threat Informed Defense. "Top att&ck techniques." (), [Online]. Available: https://github.com/center-for-threat-informed-defense/top-attack-techniques (visited on 05/23/2024).

[58] C. for Threat Informed Defense, "Top att&ck techniques - methodology," MITRE ENGENUITY, Tech. Rep. [Online]. Available: https://top-attack-techniques.mitre-engenuity.org/methodology (visited on 05/23/2024).

[59] "Top att&ck techniques." (), [Online]. Available: https://github.com/center-for-threat-informed-defense/top-attack-techniques (visited on 04/17/2024).

[60] F. Eibe, M. A. Hall, and I. H. Witten, "The weka workbench. online appendix," in "*Data Mining: Practical Machine Learning Tools and Techniques*", Fourth edition, Morgan Kaufmann, 2016.

[61] "Waikato environment for knowledge analysis." (), [Online]. Available: https://waikato.github.io/weka-wiki/ (visited on 04/16/2024).

[62] IBM. "What are naive bayes classifiers." (), [Online]. Available: https://www.ibm.com/topics/naive-bayes (visited on 05/20/2024).

[63] W. Loomis, S. Scott, T. Herr, S. A. Brackett, N. Messieh, and J. Lee, "Software supply chain security: The dataset," Atlantic Council, Tech. Rep., 2023. [Online]. Available: https://www.atlanticcouncil.org/commentary/trackers-and-data-visualizations/breaking-trust-the-dataset/ (visited on 03/20/2024).

[64] "Triage." (), [Online]. Available: https://www.tria.ge (visited on 03/22/2024).

[65] “Virustotal.” (), [Online]. Available: https://www.virustotal.com (visited on 05/18/2024).

[66] “Intezer analyze.” (), [Online]. Available: https://analyze.intezer.com/ (visited on 03/22/2024).

[67] “Mshta.exe - winbindex.” (), [Online]. Available: https://winbindex.m417z.com/?file=mshta.exe (visited on 03/22/2024).

[68] “What is .net?” (), [Online]. Available: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet (visited on 03/20/2024).

[69] E. Mesak. “Benign-net.” (), [Online]. Available: https://github.com/bormaa/Benign-NET (visited on 03/15/2024).

[70] D. Naeem and M. Brenton, “Sunburst,” MITRE, Tech. Rep., 2023. [Online]. Available: https://attack.mitre.org/software/S0559 (visited on 05/20/2024).

[71] A. Cherepanov, “Analysis of telebots’ cunning backdoor,” ESET, Tech. Rep., 2017. [Online]. Available: https://www.welivesecurity.com/2017/07/04/analysis-of-telebots-cunning-backdoor/ (visited on 05/20/2024).

Appendix A

Code

This appendix includes the Python scripts used in this project experiment. The scripts were used for the following tasks:

- Generate and filter the benign dataset.
- Perform binary differentiation using IDA Pro and Diaphora.
- Extracting the behaviors and capabilities from the new and modified functions.
- Create a malicious score based on the extracted MITRE ATT&CK techniques for each sampleset.

A.1 Benign dataset creation

To create the benign dataset from the downloaded benign .NET software described in 3, we grouped program names and sorted by version. For each program that had two or more versions, we grouped together two and two binaries in version order as tuples. These tuples were written to a database table which was used later when performing the binary diffing.

Code listing A.1: Python script for creating the benign dataset

```python
import sys
import os
import sqlite3
import argparse
from tqdm import tqdm
from hashlib import sha256
from win32api import GetFileVersionInfo, GetFileAttributes


def sort_version(version_hash_list):
    return version_hash_list[1]


def create_database(db_file):
    db_conn = sqlite3.connect(db_file)
```

```
    db = db_conn.cursor()
    db.execute('''
            CREATE TABLE IF NOT EXISTS files
            ([id] INTEGER PRIMARY KEY, [sha256] TEXT, [file_path] TEXT, [app_name] TEXT, [version] TEXT)
        ''')
    db.execute('''
            CREATE TABLE IF NOT EXISTS sample_sets
            ([id] INTEGER PRIMARY KEY, [first_file] TEXT, [second_file] TEXT)
        ''')
    db_conn.commit()
    return db_conn


def write_file_to_db(db, file_hash, file_path, app_name, version):
    query = '''INSERT INTO files (sha256, file_path, app_name, version) \
        VALUES (?, ?, ?, ?)'''
    values = (file_hash, file_path, app_name, version)
    db.execute(query, values)
    db.commit()


def write_sample_set_to_db(db, file_hash_1, file_hash_2):
    query = '''INSERT INTO sample_sets (first_file, second_file)\
        VALUES (?, ?)'''
    values = (file_hash_1, file_hash_2)
    db.execute(query, values)
    db.commit()


def get_file_properties(fname):
    """
    Read all properties of the given file return them as a dictionary.
    """
    prop_names = (
        'Comments', 'InternalName', 'ProductName',
        'CompanyName', 'LegalCopyright', 'ProductVersion',
        'FileDescription', 'LegalTrademarks', 'PrivateBuild',
        'FileVersion', 'OriginalFilename', 'SpecialBuild'
    )

    props = {
        'FixedFileInfo': None,
        'StringFileInfo': None,
        'FileVersion': None
    }

    try:
        # backslash as parm returns dictionary of numeric info corresponding to VS_FIXEDFILEINFO struc
        fixed_info = GetFileVersionInfo(fname, '\\')
        props['FixedFileInfo'] = fixed_info
        props['FileVersion'] = "%d.%d.%d.%d" % (
            fixed_info['FileVersionMS'] / 65536,
```

```
            fixed_info['FileVersionMS'] % 65536, fixed_info['FileVersionLS'] / 65536,
            fixed_info['FileVersionLS'] % 65536
        )

        # \VarFileInfo\Translation returns list of available (language, codepage)
        # pairs that can be used to retreive string info. We are using only the first pair.
        lang, codepage = GetFileVersionInfo(fname, '\\VarFileInfo\\Translation')[0]

        # any other must be of the form \StringfileInfo\%04X%04X\parm_name, middle
        # two are language/codepage pair returned from above

        str_info = {}
        for propName in prop_names:
            str_info_path = u'\\StringFileInfo\\%04X%04X\\%s' % (lang, codepage, propName)
            ## print str_info
            str_info[propName] = GetFileVersionInfo(fname, str_info_path)

        props['StringFileInfo'] = str_info
    except:
        pass

    return props["StringFileInfo"]


def sort_samples_by_name(file_info_dict, sort_key="InternalName"):
    results_dict = {}
    for key, value in file_info_dict.items():
        if value is None:
            continue
        if sort_key in value:
            internal_name = value[sort_key]
        else:
            continue
        sub_dict = {
            "FILENAME": key,
            "VERSION": value["FileVersion"],
        }
        results_dict.setdefault(internal_name, []).append(sub_dict)

    return results_dict


def main(argv=None):
    """Iterate over all files and create a data structure with attributes"""

    if argv is None:
        argv = sys.argv[1:]

    parser = argparse.ArgumentParser(description="Create a database with program versions.")
    parser.add_argument("-d", "--dir", help="Directory containing files")
```

```python
parser.add_argument("-o", "--outfile", help="Database file to write to")
args = parser.parse_args(args=argv)

db = create_database(args.outfile)

master_dict = {}

file_count = sum(len(files) for _, _, files in os.walk(args.dir))
print(f"[i] Total number of files: {file_count}")
with tqdm(total=file_count, desc="Creating database file table") as progress_bar:
    for root, dirs, files in os.walk(args.dir):
        for file in files:
            filename = os.path.join(root, file)

            # Get file properties
            file_props = get_file_properties(filename)
            if file_props is None:
                progress_bar.update(1)
                continue

            # Get file sha256sum
            with open(filename, "rb") as f:
                data = f.read()
                file_hash = sha256(data).hexdigest()

            # Get app name and version
            if "InternalName" in file_props:
                app_name = file_props["InternalName"]
            else:
                app_name = "NONE"
            if "FileVersion" in file_props:
                version = file_props["FileVersion"]
            else:
                version = "NONE"

            # Write info to db
            write_file_to_db(db, file_hash, filename, app_name, version)

            if version is not None and file_hash is not None:
                if app_name not in master_dict or (app_name in master_dict and [file_hash, version] not in master_dict[app_name]):
                    master_dict.setdefault(app_name, []).append([file_hash, version])

            progress_bar.update(1)

item_count = sum(len(value) for _, value in master_dict.items())
with tqdm(total=item_count, desc="Creating database sample set table") as progress_bar:
    for key, value in master_dict.items():
        if len(value) < 2:
            progress_bar.update(1)
            continue
```

```
            value.sort(key=sort_version)
            for i in range(1, len(value)):
                if value[i-1] != value[i]:
                    write_sample_set_to_db(db, value[i-1][0], value[i][0])
                progress_bar.update(1)
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

A.2 Binary differentiation

The binary differentiation was performed using IDA Pro and Diaphora in the script below. The script fetches each sampleset, creates an IDA database for each and then uses the Diaphora library to perform the differentiation. The result is written to a SQLite database file for each sampleset, where we can extract the unmatched functions and functions with a similiarity ratio between 0 and 1.

The second script is used to filter out samplesets that are equal or too different to be versions of the same program.

Code listing A.2: Python script for performing the binary differentiation

```
import sys
import os
import argparse
import sqlite3
import subprocess
import logging
from time import sleep

# Set program paths
diaphora_path = "C:/Users/user/Downloads/diaphora-3.1.2/diaphora/diaphora.py"
ida_log_file = "C:\\Users\\user\\Desktop\\ida.log"
db_outdir = "C:\\Users\\user\\Desktop\\analysis\\diaphora_results\\databases\\"
diff_outdir = "C:\\Users\\user\\Desktop\\analysis\\diaphora_results\\diff_results\\"


def check_file_size(file_path):
    return os.path.getsize(file_path)


def create_diaphora_db(bin_path, export_path):
    os.environ["DIAPHORA_EXPORT_FILE"] = export_path
    os.environ["DIAPHORA_AUTO"] = "1"
    os.environ["DIAPHORA_USE_DECOMPILER"] = "0"
    os.environ["DIAPHORA_PROFILE"] = "1"
    os.environ["DIAPHORA_DEBUG"] = "1"
    env = os.environ.copy()
    p = subprocess.Popen(
```

```
        ["ida64.exe", "-B", "-LC:\\Users\\user\\Desktop\\ida.log", f"-S{diaphora_path}", bin_path], shell=False,
        env=env)
    p.wait()
    sleep(0.1)

    # Clean up environment variables
    os.environ.pop("DIAPHORA_PROFILE")
    os.environ.pop("DIAPHORA_DEBUG")
    os.environ.pop("DIAPHORA_EXPORT_FILE")
    os.environ.pop("DIAPHORA_AUTO")

    return p.returncode


def do_diaphora_diff(db1, db2, diaphora_outfile):
    os.environ["DIAPHORA_DB1"] = db1
    os.environ["DIAPHORA_DB2"] = db2
    os.environ["DIAPHORA_AUTO"] = "1"
    os.environ["DIAPHORA_AUTO_DIFF"] = "1"
    os.environ["DIAPHORA_DIFF_OUT"] = diaphora_outfile
    os.environ["DIAPHORA_PROFILE"] = "1"
    os.environ["DIAPHORA_DEBUG"] = "1"
    env = os.environ.copy()
    p = subprocess.Popen(["ida64.exe", "-A", f"-L{ida_log_file}", f"-S{diaphora_path}"], shell=False, env=env)
    p.wait()
    sleep(0.1)

    # Clean up environment variables
    os.environ.pop("DIAPHORA_DB1")
    os.environ.pop("DIAPHORA_DB2")
    os.environ.pop("DIAPHORA_AUTO")
    os.environ.pop("DIAPHORA_AUTO_DIFF")
    os.environ.pop("DIAPHORA_DIFF_OUT")
    os.environ.pop("DIAPHORA_USE_DECOMPILER")
    os.environ.pop("DIAPHORA_PROFILE")
    os.environ.pop("DIAPHORA_DEBUG")

    return p.returncode


def main(argv=None):
    if argv is None:
        argv = sys.argv[1:]

    parser = argparse.ArgumentParser(description="Diff files from sqlite db")
    parser.add_argument("-f", "--file_db", help="sqlite db containing benign file data")
    args = parser.parse_args(args=argv)

    logger = logging.getLogger(__name__)
    logging.basicConfig(filename="./benign_differ.log", encoding="utf-8", level=logging.DEBUG)
```

```

    db_conn = sqlite3.connect(args.file_db)
    db = db_conn.cursor()
    diff_list_query = '''SELECT * FROM sample_sets'''
    file_info_query = '''SELECT file_path, app_name, version, sha256 FROM files WHERE sha256 = ?'''

    db.execute(diff_list_query)
    diff_list = db.fetchall()

    for sample_set in diff_list:
        db.execute(file_info_query, (sample_set[1],))
        fpath_1, appname_1, ver_1, hash_1 = db.fetchone()

        f_size = check_file_size(fpath_1)
        if f_size < 100000:
            continue

        db.execute(file_info_query, (sample_set[2],))
        fpath_2, appname_2, ver_2, hash_2 = db.fetchone()

        diff_db_1 = fpath_1 + ".sqlite"
        diff_db_2 = fpath_2 + ".sqlite"
        diff_export = diff_outdir + f"{appname_1}_{hash_1}-{hash_2}.diaphora"

        logger.info("Diffing %s version %s vs %s", appname_1, ver_1, ver_2)
        logger.info("File1=%s | File2=%s", os.path.basename(fpath_1), os.path.basename(fpath_2))

        rescode = create_diaphora_db(fpath_1, diff_db_1)
        if rescode != 0:
            logger.warning("Creating first database failed with code %s", rescode)
            continue
        rescode = create_diaphora_db(fpath_2, diff_db_2)
        if rescode != 0:
            logger.warning("Creating second database failed with code %s", rescode)
            continue
        rescode = do_diaphora_diff(diff_db_1, diff_db_2, diff_export)
        if rescode != 0:
            logger.warning("Diffing failed with code %s", rescode)
            continue

    return 0


if __name__ == "__main__":
    sys.exit(main())
```

A.3 Benign dataset filtering

After creating and conducting the binary differentiation, we filtered the benign samplesets in two stages. First, we removed the samplesets where both binaries where equal or too different to be versions of the same program. Secondly, we retrived the VirusTotal analysis statistics and removed those with many malicious verdicts. These stages were completed using the Python scripts below.

Code listing A.3: Python script for filtering out benign samplesets that have no similarities or no differences

```python
import sys
import os
import sqlite3


def get_similarity(db_path):
    conn = sqlite3.connect(db_path)
    conn.row_factory = lambda cursor, row: row[0]
    c = conn.cursor()
    c.execute("SELECT count(*) FROM unmatched")
    cnt_unmatched = c.fetchone()

    c.execute("SELECT ratio FROM results")
    ratio_list = c.fetchall()
    c.close()

    sum_ratio = 0
    for ratio in ratio_list:
        sum_ratio += float(ratio)
    cnt_total_functions = cnt_unmatched + len(ratio_list)
    similarity_percent = sum_ratio / cnt_total_functions

    return similarity_percent


def create_output_dirs():
    source_dir = "C:\\Users\\user\\Desktop\\analysis\\benign\\diaphora_results\\diff_results"
    exact_match = os.path.join(source_dir, "exact_match")
    if not os.path.exists(exact_match):
        os.mkdir(exact_match)
    partial_match = os.path.join(source_dir, "partial_match")
    if not os.path.exists(partial_match):
        os.mkdir(partial_match)
    no_match = os.path.join(source_dir, "no_match")
    if not os.path.exists(no_match):
        os.mkdir(no_match)
    return exact_match, partial_match, no_match


def main(argv=None):
    if argv is None:
        argv = sys.argv[1:]

    exact_match, partial_match, no_match = create_output_dirs()

    for filename in os.listdir(argv[0]):
        file = os.path.join(argv[0], filename)
```

```
        if os.path.isfile(file) and os.path.getsize(file):
            sim_score = get_similarity(file)
            print(f"{sim_score} ({filename})")

            if sim_score < 0.10:
                try:
                    os.rename(file, os.path.join(no_match, filename))
                except FileNotFoundError:
                    print("File Not found: ", file)
                    continue
            elif sim_score == 1.0:
                try:
                    os.rename(file, os.path.join(exact_match, filename))
                except FileNotFoundError:
                    print("File Not found: ", file)
                    continue
            else:
                try:
                    os.rename(file, os.path.join(partial_match, filename))
                except FileNotFoundError:
                    print("File Not found: ", file)
                    continue


if __name__ == "__main__":
    sys.exit(main())
```

Code listing A.4: Python script for fetching VirusTotal analysis stats

```
from tqdm import tqdm
import requests
import os
import json
import sqlite3
import sys
import re


api_key = ""

def get_vt_analysis(id):

    url = f"https://www.virustotal.com/api/v3/files/{id}"

    headers = {
        "accept": "application/json",
        "x-apikey": api_key
    }

    response = requests.get(url, headers=headers)
    if response.status_code != 200:
        print(f"VT query error: {response.content} \n{url}")
        exit(1)

    data = response.json().get("data")
```

```
    mal_score = data["attributes"]["last_analysis_stats"]
    return mal_score


def get_hashes(db_path):
    conn = sqlite3.connect(db_path)
    db = conn.cursor()
    db.execute("SELECT diff_db FROM fileinfo")
    diff_db_list = db.fetchall()
    conn.close()
    return diff_db_list


def create_vt_table(db_path):
    conn = sqlite3.connect(db_path)
    db = conn.cursor()
    # create table
    db.execute("CREATE TABLE IF NOT EXISTS vt_stats (id integer PIRMARY KEY,
     diff_db text, hash1_vt_mal integer, hash1_vt_sus integer, hash2_vt_mal integer,
      hash2_vt_sus integer)")
    conn.commit()
    return conn


def insert_vt_values(db_conn, diff_db, vt_mal1, vt_sus1, vt_mal2, vt_sus2):
    db = db_conn.cursor()
    query = "INSERT INTO vt_stats(diff_db, hash1_vt_mal, hash1_vt_sus, hash2_vt_mal
     , hash2_vt_sus) VALUES(?,?,?,?,?)"
    values = (diff_db, vt_mal1, vt_sus1, vt_mal2, vt_sus2)
    db.execute(query, values)
    db_conn.commit()


def find_hashes(string):
    pattern = r"[a-f0-9]{64}-[a-f0-9]{64}"
    matches = re.findall(pattern, string)[0].split("-")
    return matches



def main(argv=None):
    if len(sys.argv) > 1:
        argv = sys.argv[1:]
    sql_db = argv[0]
    diff_filename_list = get_hashes(sql_db)
    db_conn = create_vt_table(sql_db)
    with tqdm(total=len(diff_filename_list)) as pbar:
        for diff_fpath, in diff_filename_list:
            hashes = find_hashes(diff_fpath)
            hash1_score = get_vt_analysis(hashes[0])
            hash2_score = get_vt_analysis(hashes[1])
            if hash1_score == -1 or hash2_score == -1:
                print("Error with: %s", diff_fpath)
                insert_vt_values(db_conn, diff_fpath, -1, -1, -1, -1)
```

```
                continue
            print("VT_mal score: ", hash1_score["malicious"], hash2_score["
    malicious"])
            insert_vt_values(db_conn, diff_fpath, hash1_score["malicious"],
    hash1_score["suspicious"], hash2_score["malicious"], hash2_score["suspicious"])
        pbar.update(1)


if __name__ == "__main__":
    sys.exit(main())
```

A.4 Behavior extraction and malicious scoring

For extracting the capabilities and behaviors from the functions that are unmatched or modified, we modified an existing script from the Capa GitHub repository [14]. This script performs the Capa analysis on a program and outputs the matched behaviors to the functions where they are found. We modified the script to save the behaviors and functions to the diffing result file, and then calculate the malicious score using a CSV file containing the MITRE Top ATT&CK technique weights.

Code listing A.5: Python script for performing behavior extraction and malicious scoring

```
#!/usr/bin/env python2
# Copyright (C) 2023 Mandiant, Inc. All Rights Reserved.
# Licensed under the Apache License, Version 2.0 (the "License");
#  you may not use this file except in compliance with the License.
# You may obtain a copy of the License at: [package root]/LICENSE.txt
# Unless required by applicable law or agreed to in writing, software distributed
    under the License
#  is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND
    , either express or implied.
# See the License for the specific language governing permissions and limitations
    under the License.
"""
show-capabilities-by-function

Invoke capa to extract the capabilities of the given sample
and emit the results grouped by function.

This is useful to identify "complex functions" - that is,
functions that implement a lot of different types of logic.

Example::

    $ python scripts/show-capabilities-by-function.py /tmp/suspicious.dll_
    function at 0x1000321A with 33 features:
      - get hostname
      - initialize Winsock library
    function at 0x10003286 with 63 features:
      - create thread
      - terminate thread
```

```
    function at 0x10003415 with 116 features:
      - write file
      - send data
      - link function at runtime
      - create HTTP request
      - get common file path
      - send HTTP request
      - connect to HTTP server
    function at 0x10003797 with 81 features:
      - get socket status
      - send data
      - receive data
      - create TCP socket
      - send data on socket
      - receive data on socket
      - act as TCP client
      - resolve DNS
      - create UDP socket
      - initialize Winsock library
      - set socket configuration
      - connect TCP socket
    ...

Copyright (C) 2023 Mandiant, Inc. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
You may obtain a copy of the License at: [package root]/LICENSE.txt
Unless required by applicable law or agreed to in writing, software distributed under the License
 is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and limitations under the License.
"""
import json
import sys
import os
import logging
import argparse
import collections
from typing import Dict, Set, Any
import sqlite3
import colorama
import csv
import re
from tqdm import tqdm
from pprint import pprint

import capa.main
import capa.rules
import capa.engine
import capa.helpers
import capa.features
import capa.exceptions
```

```python
import capa.render.json
import capa.render.default
import capa.render.utils as rutils
import capa.render.verbose
import capa.features.freeze
import capa.features.freeze.features as frzf
import capa.capabilities.common
import capa.render.result_document as rd
from capa.features.freeze import Address
from capa.features.common import OS_AUTO, FORMAT_AUTO

logger = logging.getLogger("capa.show-capabilities-by-function")


def get_unmatched_functions(db_path):
    conn = sqlite3.connect(db_path)
    conn.row_factory = lambda cursor, row: row[0]
    c = conn.cursor()
    c.execute("SELECT name FROM unmatched")
    unmatched = c.fetchall()
    conn.close()
    return unmatched


def get_partial_matched_functions(db_path):
    conn = sqlite3.connect(db_path)
    conn.row_factory = lambda cursor, row: row[0]
    c = conn.cursor()
    c.execute("SELECT name2 FROM results WHERE ratio != '1.0000000' AND bb1 != '1' AND bb2 != '1'")
    partial_functions = c.fetchall()
    conn.close()
    return partial_functions


def get_diff_binary(db_path):
    conn = sqlite3.connect(db_path)
    conn.row_factory = lambda cursor, row: row[0]
    c = conn.cursor()
    c.execute("SELECT diff_db FROM config")
    result = c.fetchone()
    diff_bin_path = "".join(ch for ch in result)
    conn.close()
    return diff_bin_path


def create_capa_db_tables(db_file):
    db_conn = sqlite3.connect(db_file)
    db = db_conn.cursor()
    db.execute('''
            CREATE TABLE IF NOT EXISTS capa_attck
            ([id] INTEGER PRIMARY KEY, [function_name] TEXT, [attck] TEXT, UNIQUE(function_name, attck))
        ''')
```

```
    db.execute('''
                CREATE TABLE IF NOT EXISTS capa_mbc
                ([id] INTEGER PRIMARY KEY, [function_name] TEXT, [mbc] TEXT, UNIQUE(function_name, mbc))
            ''')
    db.execute('''
                CREATE TABLE IF NOT EXISTS capa_capability
                ([id] INTEGER PRIMARY KEY, [function_name] TEXT, [capability] TEXT, UNIQUE(function_name, capability))
            ''')
    db_conn.commit()
    return db_conn


def write_cap_to_db(db, function_name, table, capa_type, capa_str):
    query = f"INSERT OR IGNORE INTO {table} (function_name, {capa_type}) \
        VALUES (?, ?)"
    values = (function_name, capa_str)
    db.execute(query, values)
    db.commit()


def write_results_to_database(data, db_name):
    db = create_capa_db_tables(db_name)
    for key, value in data.items():
        for tactic, techniques in value["ATTCK"].items():
            for technique in techniques:
                write_cap_to_db(db, key, "capa_attck", "attck", f"{tactic}::{technique}")
        for behavior, subbehaviors in value["MBC"].items():
            for subbehavior in subbehaviors:
                write_cap_to_db(db, key, "capa_mbc", "mbc", f"{behavior}::{subbehavior}")
        for cap, subcaps in value["CAPABILITY"].items():
            for subcap in subcaps:
                write_cap_to_db(db, key, "capa_capability", "capability", f"{cap}::{subcap}")
    db.close()


def results_database(db_name, diaphora_file, mitre_score):
    db_conn = sqlite3.connect(db_name)
    db = db_conn.cursor()
    db.execute('''CREATE TABLE IF NOT EXISTS results
                    ([id] INTEGER PRIMARY KEY, [diff_db] TEXT, [mitre_score] TEXT, UNIQUE(diff_db, mitre_score))
                ''')
    db.execute("INSERT OR IGNORE INTO results (diff_db, mitre_score) VALUES (?, ?)", (diaphora_file, mitre_score))

    db_conn.commit()
    db_conn.close()
```

```
def add_mitre_score_to_db(db_name, data):
    db_conn = sqlite3.connect(db_name)
    db = db_conn.cursor()
    # Check if table exists
    db.execute('''DROP TABLE IF EXISTS mitre_scoring''')
    db.execute('''
            CREATE TABLE IF NOT EXISTS mitre_scoring
            ([id] INTEGER PRIMARY KEY, [technique] TEXT, [weight] TEXT, [
    occurences] INTEGER)
        ''')
    db.execute('''DROP TABLE IF EXISTS mitre_final_score''')
    db.execute('''
                CREATE TABLE IF NOT EXISTS mitre_final_score
                ([id] INTEGER PRIMARY KEY, [final_score] TEXT)
            ''')

    query = '''INSERT INTO mitre_scoring (technique, weight, occurences) VALUES(?,
    ?, ?)'''
    final_score = 0
    for technique, weight, occurences in data:
        final_score += (occurences * float(weight))
        values = (technique, weight, occurences)
        db.execute(query, values)

    db.execute("INSERT INTO mitre_final_score (final_score) VALUES (?)", (
    final_score,))

    db_conn.commit()
    db_conn.close()
    return final_score


def get_technique_ids(json_data):
    t_list = []
    for key, value in json_data.items():
        for techniques in value["ATTCK"].values():
            for technique in techniques:
                t_ids = re.findall("T[0-9]{4}$", technique)
                for t_id in t_ids:
                    t_list.append(t_id)
    return t_list


def mitre_score(score_file_csv, json_results):
    """Score the json_results using the mitre scoring system"""
    with open(score_file_csv) as csvfile:
        score_chart = csv.DictReader(csvfile)
        score_id = score_chart.fieldnames[0]
        techniques = get_technique_ids(json_results)
        scoring_table = []
        for row in score_chart:
            t_id = row["Technique (ID)"]
            if t_id in techniques:
                scoring_table.append((t_id, row[score_id], techniques.count(t_id)))
```

```python
    return scoring_table


def render_meta(doc: rd.ResultDocument, result):
    result["md5"] = doc.meta.sample.md5
    result["sha1"] = doc.meta.sample.sha1
    result["sha256"] = doc.meta.sample.sha256
    result["path"] = doc.meta.sample.path


def render_capabilities(rule_meta, result):
    """
    example::
        {'CAPABILITY': {'accept command line arguments': 'host-interaction/cli',
                'allocate thread local storage (2 matches)': 'host-interaction/process',
                'check for time delay via GetTickCount': 'anti-analysis/anti-debugging/debugger-detection',
                'check if process is running under wine': 'anti-analysis/anti-emulation/wine',
                'contain a resource (.rsrc) section': 'executable/pe/section/rsrc',
                'write file (3 matches)': 'host-interaction/file-system/write'}
        }
    """
    #subrule_matches = find_subrule_matches(doc)

    result["CAPABILITY"] = {}
    capability = rule_meta.name

    result["CAPABILITY"].setdefault(rule_meta.namespace, [])
    result["CAPABILITY"][rule_meta.namespace].append(capability)


def render_attack(rule_meta, result):
    """
    example::
        {'ATT&CK': {'COLLECTION': ['Input Capture::Keylogging [T1056.001]'],
            'DEFENSE EVASION': ['Obfuscated Files or Information [T1027]',
                                'Virtualization/Sandbox Evasion::System Checks '
                                '[T1497.001]'],
            'DISCOVERY': ['File and Directory Discovery [T1083]',
                          'Query Registry [T1012]',
                          'System Information Discovery [T1082]'],
            'EXECUTION': ['Shared Modules [T1129]']}
        }
    """
    result["ATTCK"] = {}
    tactics = collections.defaultdict(set)
    if not rule_meta.attack:
        return -1
    for attack in rule_meta.attack:
        tactics[attack.tactic].add((attack.technique, attack.subtechnique, attack.id))

```

```
    for tactic, techniques in sorted(tactics.items()):
        inner_rows = []
        for technique, subtechnique, id in sorted(techniques):
            if subtechnique is None:
                inner_rows.append(f"{technique} {id}")
            else:
                inner_rows.append(f"{technique}::{subtechnique} {id}")
        result["ATTCK"].setdefault(tactic.upper(), inner_rows)


def render_mbc(rule_meta, result):
    """
    example::
        {'MBC': {'ANTI-BEHAVIORAL ANALYSIS': ['Debugger Detection::Timing/Delay Check '
                                              'GetTickCount [B0001.032]',
                                              'Emulator Detection [B0004]',
                                              'Virtual Machine Detection::Instruction '
                                              'Testing [B0009.029]',
                                              'Virtual Machine Detection [B0009]'],
         'COLLECTION': ['Keylogging::Polling [F0002.002]'],
         'CRYPTOGRAPHY': ['Encrypt Data::RC4 [C0027.009]',
                          'Generate Pseudo-random Sequence::RC4 PRGA '
                          '[C0021.004]']}
        }
    """
    result["MBC"] = {}
    objectives = collections.defaultdict(set)
    if not rule_meta.mbc:
        return -1

    for mbc in rule_meta.mbc:
        objectives[mbc.objective].add((mbc.behavior, mbc.method, mbc.id))

    for objective, behaviors in sorted(objectives.items()):
        inner_rows = []
        for behavior, method, id in sorted(behaviors):
            if method is None:
                inner_rows.append(f"{behavior} [{id}]")
            else:
                inner_rows.append(f"{behavior}::{method} [{id}]")
        result["MBC"].setdefault(objective.upper(), inner_rows)


def render_dictionary(rule_meta) -> Dict[str, Any]:
    result: Dict[str, Any] = {}
    #render_meta(rule_meta, result)
    render_attack(rule_meta, result)
    render_mbc(rule_meta, result)
    render_capabilities(rule_meta, result)
    return result


def render_matches_by_function(doc: rd.ResultDocument, extractor, diaphora_db):
```

```
    """
    like:

        function at 0x1000321a with 33 features:
          - get hostname
          - initialize Winsock library
        function at 0x10003286 with 63 features:
          - create thread
          - terminate thread
        function at 0x10003415 with 116 features:
          - write file
          - send data
          - link function at runtime
          - create HTTP request
          - get common file path
          - send HTTP request
          - connect to HTTP server
    """
    assert isinstance(doc.meta.analysis, rd.StaticAnalysis)
    functions_by_bb: Dict[Address, Address] = {}
    for finfo in doc.meta.analysis.layout.functions:
        faddress = finfo.address

        for bb in finfo.matched_basic_blocks:
            bbaddress = bb.address
            functions_by_bb[bbaddress] = faddress

    ostream = rutils.StringIO()

    matches_by_function: Dict[Any, Any] = {}

    for rule in rutils.capability_rules(doc):
        if capa.rules.Scope.FUNCTION in rule.meta.scopes:
            for addr, _ in rule.matches:
                matches_by_function[addr] = render_dictionary(rule.meta)

        elif capa.rules.Scope.BASIC_BLOCK in rule.meta.scopes:
            for addr, _ in rule.matches:
                function = functions_by_bb[addr]
                matches_by_function[function] = render_dictionary(rule.meta)
        else:
            # file scope
            pass

    # Get diaphora diffing results
    new_functions = get_unmatched_functions(diaphora_db)
    modified_functions = get_partial_matched_functions(diaphora_db)

    result: Dict[Any, Any] = {}

    if doc.meta.analysis.extractor != "DnfileFeatureExtractor":
        # Parse function names to address values
        for i in range(0, len(new_functions)):
            try:
```

```
                new_functions[i] = hex(int(new_functions[i].lstrip("sub_"), 16))
            except ValueError:
                continue
        for i in range(0, len(modified_functions)):
            try:
                modified_functions[i] = hex(int(modified_functions[i].lstrip("sub_"), 16))
            except ValueError:
                continue

        for f in doc.meta.analysis.feature_counts.functions:
            if not matches_by_function.get(f.address, {}):
                continue
            f_addr_formated = capa.render.verbose.format_address(f.address)

            if f_addr_formated in new_functions:
                ostream.writeln(f"New function at {f_addr_formated} with {f.count} features: ")
                result[f_addr_formated] = matches_by_function[f.address]
                result[f_addr_formated]["MATCH"] = "UNMATCHED"

            if f_addr_formated in modified_functions:
                ostream.writeln(f"Modified function at {f_addr_formated} with {f.count} features: ")
                result[f_addr_formated] = matches_by_function[f.address]
                result[f_addr_formated]["MATCH"] = "PARTIAL"

    else:
        for f in doc.meta.analysis.feature_counts.functions:
            if not matches_by_function.get(f.address, {}):
                continue
            func_name = str(extractor.token_cache.methods[f.address.value])

            for matching_name in new_functions:
                if matching_name in func_name or func_name in matching_name:
                    # ostream.writeln(f"New function at {func_name} ({f.address}) with {f.count} features")
                    result[func_name] = matches_by_function[f.address]
                    result[func_name]["MATCH"] = "UNMATCHED"

            for matching_name in modified_functions:
                if matching_name in func_name or func_name in matching_name:
                    # ostream.writeln(f"Modified function at {func_name} ({f.address}) with {f.count} features")
                    result[func_name] = matches_by_function[f.address]
                    result[func_name]["MATCH"] = "PARTIAL"
    # print(ostream.getvalue())
    return result


def main(argv=None):
    if argv is None:
        argv = sys.argv[1:]

```

```
    parser = argparse.ArgumentParser(description="detect capabilities in programs.")
    capa.main.install_common_args(
        parser, wanted={"format", "os", "backend", "input_file", "signatures", "rules", "tag"}
    )
    #parser.add_argument("-x", "--diaphorafile", help="path to .diaphora file")

    logger = logging.getLogger(__name__)
    logging.basicConfig(filename="./capability_extraction.log", encoding="utf-8", level=logging.DEBUG)

    args = parser.parse_args(args=argv)
    diffing_dir = args.input_file
    dir_list = os.listdir(diffing_dir)

    with tqdm(total=len(dir_list), desc="Extracting behavior from differing functions") as progress_bar:
        for filename in dir_list:
            if not filename.endswith(".diaphora"):
                continue
            # Get the necessary binary file (last version) from diaphora file
            diaphora_file = os.path.join(diffing_dir, filename)
            binary_db = get_diff_binary(diaphora_file)
            binary = os.path.splitext(binary_db)[:1]
            binary = "".join(x for x in binary)
            # Set the input binary
            args.input_file = binary

            try:
                capa.main.handle_common_args(args)
                capa.main.ensure_input_exists_from_cli(args)
                input_format = capa.main.get_input_format_from_cli(args)
                rules = capa.main.get_rules_from_cli(args)
                backend = capa.main.get_backend_from_cli(args, input_format)
                sample_path = capa.main.get_sample_path_from_cli(args, backend)
                if sample_path is None:
                    os_ = "unknown"
                else:
                    os_ = capa.loader.get_os(sample_path)
                extractor = capa.main.get_extractor_from_cli(args, input_format, backend)
            except capa.main.ShouldExitError as e:
                return e.status_code

            capabilities, counts = capa.capabilities.common.find_capabilities(rules, extractor)

            meta = capa.loader.collect_metadata(argv, args.input_file, input_format, os_, args.rules, extractor, counts)
            meta.analysis.layout = capa.loader.compute_layout(rules, extractor, capabilities)

            if capa.capabilities.common.has_file_limitation(rules, capabilities):
```

```
                # bail if capa encountered file limitation e.g. a packed binary
                # do show the output in verbose mode, though.
                if not (args.verbose or args.vverbose or args.json):
                    return capa.main.E_FILE_LIMITATION

            doc = rd.ResultDocument.from_capa(meta, rules, capabilities)

            result_dict = render_matches_by_function(doc, extractor, diaphora_file)

            results = json.dumps(result_dict)
            json_data = json.loads(results)
            json_file = os.path.splitext(diaphora_file)[0] + ".json"
            with open(json_file, "w") as j_file:
                json.dump(json_data, j_file)
            write_results_to_database(json_data, diaphora_file)
            scoring_table = mitre_score("./mitre_scores.csv", json_data)
            final_score = add_mitre_score_to_db(diaphora_file, scoring_table)
            results_database("./final_results.sqlite", diaphora_file, final_score)

            logger.info("Capa completed %s", diaphora_file)

            progress_bar.update(1)

    colorama.deinit()
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

Appendix B

Behavior identification results

This appendix includes detailed tables from the behavior identification, displaying the frequency of the MBC identifiers and the Capa capabilities for the benign and malicious dataset.

Table B.1: Complete table of MBC identifiers and number of occurrences in benign and malicious datasets

MBC identifier	Benign	Malicious
DISCOVERY::File and Directory Discovery:: [E1083]	1883	16
PROCESS::Create Process:: [C0017]	398	3
PROCESS::Suspend Thread:: [C0055]	322	7
PROCESS::Terminate Process:: [C0018]	289	3
OPERATING SYSTEM::Console:: [C0033]	247	1
FILE SYSTEM::Writes File:: [C0052]	150	2
DISCOVERY::System Information Discovery:: [E1082]	141	4
OPERATING SYSTEM::Registry::Query Registry Value [C0036.006]	131	6
CRYPTOGRAPHY::Generate Pseudo-random Sequence::Use API [C0021.003]	119	11
PROCESS::Create Thread:: [C0038]	123	5
FILE SYSTEM::Read File:: [C0051]	114	6
FILE SYSTEM::Create Directory:: [C0046]	109	1
FILE SYSTEM::Delete File:: [C0047]	101	9
FILE SYSTEM::Move File:: [C0063]	86	2
DEFENSE EVASION::Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]	70	13

OPERATING SYSTEM::Registry::Set Registry Key [C0036.001]	75	3
FILE SYSTEM::Copy File:: [C0045]	77	0
IMPACT::Clipboard Modification:: [E1510]	74	0
DATA::Encode Data::Base64 [C0026.001]	58	12
COMMUNICATION::HTTP Communication::Send Request [C0002.003]	57	3
FILE SYSTEM::Set File Attributes:: [C0050]	52	4
FILE SYSTEM::Get File Attributes:: [C0049]	44	1
DATA::Decode Data::Base64 [C0053.001]	33	9
OPERATING SYSTEM::Registry::Query Registry Key [C0036.005]	34	2
COMMUNICATION::HTTP Communication::Get Response [C0002.017]	12	11
OPERATING SYSTEM::Registry::Delete Registry Value [C0036.007]	22	1
FILE SYSTEM::Delete Directory:: [C0048]	18	0
COMMUNICATION::HTTP Communication::Download URL [C0002.006]	15	0
CRYPTOGRAPHY::Cryptographic Hash::MD5 [C0029.001]	14	1
OPERATING SYSTEM::Registry::Delete Registry Key [C0036.002]	15	0
COMMUNICATION::HTTP Communication::Create Request [C0002.012]	14	0
ANTI-BEHAVIORAL ANALYSIS::Debugger Detection::WudfIsAnyDebuggerPresent [B0001.031]	12	0
ANTI-BEHAVIORAL ANALYSIS::Debugger Detection::CheckRemoteDebuggerPresent [B0001.002]	12	0
PROCESS::Create Mutex:: [C0042]	10	0
PERSISTENCE::Registry Run Keys / Startup Folder:: [F0012]	10	0
EXECUTION::Command and Scripting Interpreter:: [E1059]	9	0
COMMUNICATION::Socket Communication::Receive Data [C0001.006]	8	0
COMMUNICATION::Socket Communication::Create TCP Socket [C0001.011]	6	0
COMMUNICATION::Socket Communication::Create UDP Socket [C0001.010]	5	1
CRYPTOGRAPHY::Cryptographic Hash::SHA1 [C0029.002]	5	0

CRYPTOGRAPHY::Encrypt Data:: [C0027]	5	0
COMMUNICATION::Socket Communication::Send Data [C0001.007]	5	0
DATA::Compress Data:: [C0024]	3	2
COMMUNICATION::DNS Communication::Resolve [C0011.001]	3	1
COMMUNICATION::Socket Communication::TCP Client [C0001.008]	3	0
COMMUNICATION::Socket Communication::Start TCP Server [C0001.005]	3	0
CRYPTOGRAPHY::Cryptographic Hash::SHA256 [C0029.003]	3	0
OPERATING SYSTEM::Environment Variable::Set Variable [C0034.001]	2	0
COLLECTION::Screen Capture::WinAPI [E1113.m01]	1	0
DISCOVERY::Code Discovery::Inspect Section Memory Permissions [B0046.002]	1	0
ANTI-STATIC ANALYSIS::Executable Code Obfuscation::Argument Obfuscation [B0032.020]	0	1
ANTI-STATIC ANALYSIS::Executable Code Obfuscation::Stack Strings [B0032.017]	0	1
DATA::Encode Data::XOR [C0026.002]	0	1
COMMUNICATION::HTTP Communication::Read Header [C0002.014]	0	1
DATA::Checksum::CRC32 [C0032.001]	0	1

Table B.2: Complete table of Capa capabilities and number of occurrences in benign and malicious datasets

Capa capability	**Benign**	**Malicious**
runtime::unmanaged call	1092	7
host-interaction/file-system/exists::check if file exists	1018	7
host-interaction/file-system/files/list::enumerate files in .NET	579	2
data-manipulation/regex::find data using regex in .NET	417	7
host-interaction/process/create::create process on Windows	398	3
host-interaction/thread/suspend::suspend thread	322	7

host-interaction/process/terminate::terminate process	289	3
host-interaction/file-system/exists::check if directory exists	274	2
host-interaction/console::manipulate console buffer	245	1
host-interaction/memory::manipulate unmanaged memory in .NET	231	0
host-interaction/file-system::check file extension in .NET	224	0
data-manipulation/xml::load XML in .NET	159	19
host-interaction/file-system/write::write file on Windows	150	2
host-interaction/registry::query or enumerate registry value	131	6
host-interaction/thread/create::create thread	123	5
host-interaction/file-system::get common file path	121	5
host-interaction/os/version::get OS version in .NET	123	3
data-manipulation/prng::generate random numbers in .NET	113	11
host-interaction/file-system/read::read file on Windows	114	6
load-code/dotnet::load .NET assembly	116	1
host-interaction/file-system/delete::delete file	101	9
host-interaction/file-system/create::create directory	109	1
collection/database/sql::reference SQL statements	14	94
host-interaction/file-system/meta::get file size	92	1
host-interaction/file-system/move::move file	86	2
host-interaction/registry/create::set registry value	75	3
host-interaction/file-system/copy::copy file	77	0
host-interaction/environment-variable::query environment variable	73	3
host-interaction/clipboard::write clipboard data	74	0
host-interaction/file-system/meta::get file version info	73	1
data-manipulation/encoding/base64::encode data using Base64	58	12
host-interaction/session::get session user name	56	5
host-interaction/os/hostname::get hostname	59	1
host-interaction/file-system/meta::set file attributes	52	4

executable/resource::access .NET resource	53	2
communication/authentication::manipulate network credentials in .NET	55	0
load-code/dotnet::generate method via reflection in .NET	51	0
communication/http/client::send HTTP request	45	3
host-interaction/file-system/meta::get file attributes	44	1
data-manipulation/encoding/base64::decode data using Base64 in .NET	33	9
host-interaction/registry::query or enumerate registry key	34	2
collection::get geographical location	31	1
host-interaction/process/list::find process by PID	29	0
host-interaction/process/list::find process by name	26	0
collection::save image in .NET	24	0
host-interaction/registry/delete::delete registry value	22	1
communication/http/client::read data from Internet	12	11
host-interaction/gui::enumerate gui resources	20	0
host-interaction/wmi::access WMI data in .NET	14	5
host-interaction/clipboard::read clipboard data	19	0
host-interaction/file-system/delete::delete directory	18	0
data-manipulation/compression::create zip archive in .NET	17	0
communication/http/client::download URL	15	0
host-interaction/file-system::generate random filename in .NET	15	0
host-interaction/process/list::enumerate processes	14	1
host-interaction/registry/delete::delete registry key	15	0
data-manipulation/hashing/md5::hash data with MD5	14	1
communication/http/client::create HTTP request	14	0
host-interaction/file-system::set current directory	13	0
data-manipulation/database/sql::execute SQLite statement in .NET	13	0
data-manipulation/encoding/url::decode data using URL encoding	12	0
communication/http/client::send request in .NET	12	0

load-code/dotnet::invoke .NET assembly method	12	0
anti-analysis/anti-debugging/debugger-detection::check for debugger via API	12	0
host-interaction/hardware/keyboard::send keystrokes	12	0
host-interaction/file-system/move::move directory	11	0
communication/http::set web proxy in .NET	11	0
host-interaction/mutex::create mutex	10	0
persistence/registry/run::persist via Run registry key	10	0
host-interaction/cli::accept command line arguments	9	0
host-interaction/thread/task::execute via asynchronous task in .NET	9	0
communication/socket/receive::receive data on socket	8	0
communication/http/client::send data to Internet	4	3
communication/socket/tcp::create TCP socket	6	0
host-interaction/clipboard::clear clipboard data	6	0
communication/socket/udp/send::create UDP socket	5	1
host-interaction/thread/timer::execute via timer in .NET	6	0
data-manipulation/prng::generate random bytes in .NET	6	0
host-interaction/hardware/storage::get disk information	6	0
data-manipulation/encryption/dpapi::encrypt data using DPAPI	5	0
host-interaction/process::get process image filename	5	0
host-interaction/process/terminate::terminate process by name in .NET	5	0
communication/http::set HTTP User-Agent in .NET	5	0
data-manipulation/hashing/sha1::hash data using SHA1	5	0
communication/socket/send::send data on socket	5	0
data-manipulation/compression::compress data using GZip in .NET	3	2
host-interaction/hardware/cpu::get number of processors	5	0
communication/http::get system web proxy	4	1

persistence/service::persist via Windows service	5	0
communication/http::set HTTP cookie	4	0
communication/dns::resolve DNS	3	1
communication/tcp/client::act as TCP client	3	0
host-interaction/hardware/storage::get disk size	3	0
persistence::create shortcut via IShellLink	0	3
host-interaction/network/interface::get networking interfaces	2	1
communication/tcp/serve::start TCP server	3	0
data-manipulation/hashing/sha256::hash data using SHA256	3	0
host-interaction/gui::display service notification message box	3	0
host-interaction/console::manipulate console window	2	0
host-interaction/process/modify::acquire debug privileges	2	0
collection/network::get MAC address in .NET	1	1
data-manipulation/json::deserialize JSON in .NET	2	0
data-manipulation/json::serialize JSON in .NET	2	0
persistence/scheduled-tasks::schedule task via at	2	0
host-interaction/environment-variable::set environment variable	2	0
collection/webcam::capture webcam image	0	2
linking/runtime-linking::link function at runtime on Windows	0	2
data-manipulation/encoding/xor::encode data using XOR	0	1
communication/http::read HTTP header	0	1
host-interaction/network/domain::get domain information	0	1
data-manipulation/checksum/crc32::hash data with CRC32	0	1
linking/runtime-linking::get kernel32 base address	0	1
load-code/dotnet/csharp::compile CSharp in .NET	1	0
anti-analysis/obfuscation/string/stackstring::contain obfuscated stackstrings	0	1
host-interaction/file-system::reference absolute stream path on Windows	1	0
runtime/dotnet::unmanaged call via dynamic PInvoke in .NET	1	0
load-code/pe::inspect section memory permissions	1	0

collection/screenshot::capture screenshot	1	0
host-interaction/clipboard::check clipboard data	1	0
load-code/dotnet::compile .NET assembly	1	0
load-code/pe::resolve function by parsing PE exports	0	1
persistence/scheduled-tasks::schedule task via schtasks	0	0

Appendix C

Malicious scoring weights

This appendix includes the weights used in the malicious scoring, where the malicious score was calculated by multiplying these weights with the number of occurrences of the techniques. This table is generated from the MITRE Top ATT&CK technique spreadsheet [59] as described in chapter 3.

Table C.1: All weights from the MITRE Top 10 ATT&CK techniques

Total Top Att&ck Score	Technique (ID)	Technique (Name)	Sub-technique (Name)
2.914286	T1059	Command and Scripting Interpreter	NaN
2.183333	T1047	Windows Management Instrumentation	NaN
2.114286	T1053	Scheduled Task/Job	NaN
1.945238	T1055	Process Injection	NaN
1.880952	T1218	Signed Binary Proxy Execution	NaN
1.826190	T1574	Hijack Execution Flow	NaN
1.804762	T1562	Impair Defenses	NaN
1.766667	T1543	Create or Modify System Process	NaN
1.619048	T1036	Masquerading	NaN
1.604762	T1112	Modify Registry	NaN
1.592243	T1021	Remote Services	NaN
1.586262	T1105	Ingress Tool Transfer	NaN
1.494314	T1027	Obfuscated Files or Information	NaN
1.460332	T1003	OS Credential Dumping	NaN

1.411905	T1095	Non-Application Layer Protocol	NaN
1.392857	T1090	Proxy	NaN
1.357723	T1078	Valid Accounts	NaN
1.311811	T1204	User Execution	NaN
1.306383	T1548	Abuse Elevation Control Mechanism	NaN
1.296056	T1070	Indicator Removal on Host	NaN
1.242857	T1557.003	Adversary-in-the-Middle	DHCP Spoofing
1.181680	T1074	Data Staged	NaN
1.147619	T1559.003	Inter-Process Communication	XPC Services
1.128571	T1564.010	Hide Artifacts	Process Argument Spoofing
1.083195	T1190	Exploit Public-Facing Application	NaN
1.071094	T1569	System Services	NaN
1.069018	T1552	Unsecured Credentials	NaN
0.989403	T1547	Boot or Logon Autostart Execution	NaN
0.952785	T1106	Native API	NaN
0.947619	T1059.001	Command and Scripting Interpreter	PowerShell
0.935646	T1219	Remote Access Software	NaN
0.923648	T1068	Exploitation for Privilege Escalation	NaN
0.909524	T1003.001	OS Credential Dumping	LSASS Memory
0.871188	T1110	Brute Force	NaN
0.870046	T1564	Hide Artifacts	NaN
0.861174	T1072	Software Deployment Tools	NaN
0.807143	T1484	Domain Policy Modification	NaN
0.781106	T1071	Application Layer Protocol	NaN
0.772202	T1557	Adversary-in-the-Middle	NaN
0.745593	T1098	Account Manipulation	NaN
0.730952	T1210	Exploitation of Remote Services	NaN
0.719175	T1570	Lateral Tool Transfer	NaN
0.685714	T1021.001	Remote Services	Remote Desktop Protocol

0.685644	T1197	BITS Jobs	NaN
0.684502	T1560	Archive Collected Data	NaN
0.680675	T1082	System Information Discovery	NaN
0.669344	T1136	Create Account	NaN
0.669048	T1561.002	Disk Wipe	Disk Structure Wipe
0.665455	T1546	Event Triggered Execution	NaN
0.661905	T1053.005	Scheduled Task/Job	Scheduled Task
0.647619	T1530	Data from Cloud Storage Object	NaN
0.638095	T1021.002	Remote Services	SMB/Windows Admin Shares
0.623227	T1490	Inhibit System Recovery	NaN
0.622111	T1056	Input Capture	NaN
0.616667	T1528	Steal Application Access Token	NaN
0.603350	T1222	File and Directory Permissions Modification	NaN
0.595238	T1543.003	Create or Modify System Process	Windows Service
0.592951	T1489	Service Stop	NaN
0.591802	T1559	Inter-Process Communication	NaN
0.590476	T1003.002	OS Credential Dumping	Security Account Manager
0.590328	T1566	Phishing	NaN
0.588021	T1203	Exploitation for Client Execution	NaN
0.576190	T1078.004	Valid Accounts	Cloud Accounts
0.567839	T1140	Deobfuscate/Decode Files or Information	NaN
0.559524	T1565.003	Data Manipulation	Runtime Data Manipulation
0.558090	T1571	Non-Standard Port	NaN
0.557143	T1563.002	Remote Service Session Hijacking	RDP Hijacking
0.555269	T1012	Query Registry	NaN
0.554918	T1558	Steal or Forge Kerberos Tickets	NaN
0.552381	T1070.001	Indicator Removal on Host	Clear Windows Event Logs

0.547619	T1003.003	OS Credential Dumping	NTDS
0.538095	T1547.015	Boot or Logon Autostart Execution	Login Items
0.537410	T1102	Web Service	NaN
0.533333	T1048.003	Exfiltration Over Alternative Protocol	Exfiltration Over Unencrypted/Obfuscated Non-C2 Protocol
0.528571	T1133	External Remote Services	NaN
0.523810	T1059.005	Command and Scripting Interpreter	Visual Basic
0.523810	T1548.001	Abuse Elevation Control Mechanism	Setuid and Setgid
0.520876	T1048	Exfiltration Over Alternative Protocol	NaN
0.519048	T1021.006	Remote Services	Windows Remote Management
0.519048	T1548.002	Abuse Elevation Control Mechanism	Bypass User Account Control
0.512381	T1555	Credentials from Password Stores	NaN
0.504762	T1542.005	Pre-OS Boot	TFTP Boot
0.500000	T1602.002	Data from Configuration Repository	Network Device Configuration Dump
0.483947	T1482	Domain Trust Discovery	NaN
0.480952	T1602.001	Data from Configuration Repository	SNMP (MIB Dump)
0.479206	T1565	Data Manipulation	NaN
0.476190	T1574.001	Hijack Execution Flow	DLL Search Order Hijacking
0.476190	T1213	Data from Information Repositories	NaN
0.475858	T1087	Account Discovery	NaN
0.471711	T1080	Taint Shared Content	NaN
0.471429	T1070.002	Indicator Removal on Host	Clear Linux or Mac System Logs
0.471429	T1569.001	System Services	Launchctl
0.471429	T1213.001	Data from Information Repositories	Confluence
0.471429	T1213.002	Data from Information Repositories	Sharepoint
0.466667	T1021.005	Remote Services	VNC

0.462302	T1505	Server Software Component	NaN
0.461905	T1136.003	Create Account	Cloud Account
0.461905	T1601	Modify System Image	NaN
0.458370	T1091	Replication Through Removable Media	NaN
0.457143	T1212	Exploitation for Credential Access	NaN
0.452592	T1563	Remote Service Session Hijacking	NaN
0.452381	T1557.002	Adversary-in-the-Middle	ARP Cache Poisoning
0.452381	T1558.002	Steal or Forge Kerberos Tickets	Silver Ticket
0.452381	T1114	Email Collection	NaN
0.450483	T1127	Trusted Developer Utilities Proxy Execution	NaN
0.447619	T1021.003	Remote Services	Distributed Component Object Model
0.445238	T1539	Steal Web Session Cookie	NaN
0.442857	T1547.006	Boot or Logon Autostart Execution	Kernel Modules and Extensions
0.442857	T1036.003	Masquerading	Rename System Utilities
0.442857	T1059.007	Command and Scripting Interpreter	JavaScript/JScript
0.442583	T1497	Virtualization/Sandbox Evasion	NaN
0.441743	T1211	Exploitation for Defense Evasion	NaN
0.438095	T1552.001	Unsecured Credentials	Credentials In Files
0.438095	T1003.005	OS Credential Dumping	Cached Domain Credentials
0.438095	T1078.003	Valid Accounts	Local Accounts
0.438095	T1110.003	Brute Force	Password Spraying
0.433333	T1552.004	Unsecured Credentials	Private Keys
0.433333	T1537	Transfer Data to Cloud Account	NaN
0.430952	T1602	Data from Configuration Repository	NaN
0.424444	T1189	Drive-by Compromise	NaN
0.423810	T1204.002	User Execution	Malicious File
0.419725	T1542	Pre-OS Boot	NaN

0.419048	T1505.002	Server Software Component	Transport Agent
0.416995	T1134	Access Token Manipulation	NaN
0.414286	T1136.002	Create Account	Domain Account
0.414286	T1048.002	Exfiltration Over Alternative Protocol	Exfiltration Over Asymmetric Encrypted Non-C2 Protocol
0.411905	T1601.001	Modify System Image	Patch System Image
0.411905	T1601.002	Modify System Image	Downgrade System Image
0.410946	T1046	Network Service Scanning	NaN
0.409524	T1557.001	Adversary-in-the-Middle	LLMNR/NBT-NS Poisoning and SMB Relay
0.409524	T1562.006	Impair Defenses	Indicator Blocking
0.409524	T1098.001	Account Manipulation	Additional Cloud Credentials
0.405481	T1572	Protocol Tunneling	NaN
0.404762	T1021.004	Remote Services	SSH
0.404762	T1563.001	Remote Service Session Hijacking	SSH Hijacking
0.404762	T1053.002	Scheduled Task/Job	At (Windows)
0.404762	T1059.006	Command and Scripting Interpreter	Python
0.404762	T1542.001	Pre-OS Boot	System Firmware
0.400000	T1003.004	OS Credential Dumping	LSA Secrets
0.400000	T1053.003	Scheduled Task/Job	Cron
0.397619	T1565.001	Data Manipulation	Stored Data Manipulation
0.395238	T1053.001	Scheduled Task/Job	At (Linux)
0.395238	T1218.011	Signed Binary Proxy Execution	Rundll32
0.395238	T1556	Modify Authentication Process	NaN
0.390476	T1558.003	Steal or Forge Kerberos Tickets	Kerberoasting
0.390476	T1552.002	Unsecured Credentials	Credentials in Registry
0.390476	T1218.012	Signed Binary Proxy Execution	Verclsid
0.390284	T1137	Office Application Startup	NaN

0.385714	T1218.005	Signed Binary Proxy Execution	Mshta
0.385714	T1003.006	OS Credential Dumping	DCSync
0.385714	T1078.002	Valid Accounts	Domain Accounts
0.382257	T1553	Subvert Trust Controls	NaN
0.381423	T1040	Network Sniffing	NaN
0.380952	T1574.010	Hijack Execution Flow	Services File Permissions Weakness
0.379893	T1083	File and Directory Discovery	NaN
0.376190	T1059.003	Command and Scripting Interpreter	Windows Command Shell
0.374096	T1005	Data from Local System	NaN
0.371429	T1562.001	Impair Defenses	Disable or Modify Tools
0.369090	T1485	Data Destruction	NaN
0.366667	T1136.001	Create Account	Local Account
0.366667	T1071.004	Application Layer Protocol	DNS
0.361905	T1543.002	Create or Modify System Process	Systemd Service
0.361905	T1556.001	Modify Authentication Process	Domain Controller Authentication
0.361905	T1071.001	Application Layer Protocol	Web Protocols
0.361905	T1546.003	Event Triggered Execution	Windows Management Instrumentation Event Subscription
0.361905	T1558.004	Steal or Forge Kerberos Tickets	AS-REP Roasting
0.361905	T1059.004	Command and Scripting Interpreter	Unix Shell
0.359524	T1599.001	Network Boundary Bridging	Network Address Translation Traversal
0.358990	T1113	Screen Capture	NaN
0.357143	T1110.001	Brute Force	Password Guessing
0.357143	T1573.002	Encrypted Channel	Asymmetric Cryptography
0.354762	T1187	Forced Authentication	NaN
0.354762	T1599	Network Boundary Bridging	NaN
0.352381	T1550.001	Use Alternate Authentication Material	Application Access Token

0.352381	T1505.003	Server Software Component	Web Shell
0.352381	T1574.008	Hijack Execution Flow	Path Interception by Search Order Hijacking
0.352381	T1574.009	Hijack Execution Flow	Path Interception by Unquoted Path
0.352381	T1542.003	Pre-OS Boot	Bootkit
0.352381	T1547.011	Boot or Logon Autostart Execution	Plist Modification
0.351638	T1195	Supply Chain Compromise	NaN
0.347619	T1562.004	Impair Defenses	Disable or Modify System Firewall
0.347619	T1218.007	Signed Binary Proxy Execution	Msiexec
0.347619	T1566.001	Phishing	Spearphishing Attachment
0.347619	T1574.007	Hijack Execution Flow	Path Interception by PATH Environment Variable
0.342984	T1525	Implant Container Image	NaN
0.342857	T1087.002	Account Discovery	Domain Account
0.342857	T1221	Template Injection	NaN
0.342780	T1037	Boot or Logon Initialization Scripts	NaN
0.340476	T1111	Two-Factor Authentication Interception	NaN
0.338095	T1036.005	Masquerading	Match Legitimate Name or Location
0.338095	T1052	Exfiltration Over Physical Medium	NaN
0.338095	T1052.001	Exfiltration Over Physical Medium	Exfiltration over USB
0.338095	T1574.005	Hijack Execution Flow	Executable Installer File Permissions Weakness
0.338095	T1059.008	Command and Scripting Interpreter	Network Device CLI
0.338095	T1542.004	Pre-OS Boot	ROMMONkit
0.338095	T1552.005	Unsecured Credentials	Cloud Instance Metadata API
0.333333	T1110.004	Brute Force	Credential Stuffing

0.333333	T1195.002	Supply Chain Compromise	Compromise Software Supply Chain
0.333333	T1550.002	Use Alternate Authentication Material	Pass the Hash
0.328571	T1098.004	Account Manipulation	SSH Authorized Keys
0.328571	T1556.004	Modify Authentication Process	Network Device Authentication
0.324360	T1176	Browser Extensions	NaN
0.323810	T1195.001	Supply Chain Compromise	Compromise Software Dependencies and Development Tools
0.323810	T1547.004	Boot or Logon Autostart Execution	Winlogon Helper DLL
0.323810	T1218.003	Signed Binary Proxy Execution	CMSTP
0.320306	T1119	Automated Collection	NaN
0.319048	T1003.008	OS Credential Dumping	/etc/passwd and /etc/shadow
0.319048	T1552.006	Unsecured Credentials	Group Policy Preferences
0.319048	T1547.001	Boot or Logon Autostart Execution	Registry Run Keys / Startup Folder
0.319048	T1569.002	System Services	Service Execution
0.319048	T1218.004	Signed Binary Proxy Execution	InstallUtil
0.319048	T1505.001	Server Software Component	SQL Stored Procedures
0.314286	T1048.001	Exfiltration Over Alternative Protocol	Exfiltration Over Symmetric Encrypted Non-C2 Protocol
0.314286	T1204.001	User Execution	Malicious Link
0.314286	T1218.001	Signed Binary Proxy Execution	Compiled HTML File
0.314286	T1222.001	File and Directory Permissions Modification	Windows File and Directory Permissions Modification
0.314286	T1222.002	File and Directory Permissions Modification	Linux and Mac File and Directory Permissions Modification
0.311905	T1495	Firmware Corruption	NaN
0.311905	T1550	Use Alternate Authentication Material	NaN

0.310246	T1041	Exfiltration Over C2 Channel	NaN
0.309524	T1003.007	OS Credential Dumping	Proc Filesystem
0.309524	T1098.003	Account Manipulation	Add Office 365 Global Administrator Role
0.309524	T1218.009	Signed Binary Proxy Execution	Regsvcs/Regasm
0.308781	T1185	Browser Session Hijacking	NaN
0.304762	T1567	Exfiltration Over Web Service	NaN
0.304762	T1548.003	Abuse Elevation Control Mechanism	Sudo and Sudo Caching
0.304762	T1078.001	Valid Accounts	Default Accounts
0.302381	T1606	Forge Web Credentials	NaN
0.301784	T1486	Data Encrypted for Impact	NaN
0.300443	T1018	Remote System Discovery	NaN
0.300000	T1562.002	Impair Defenses	Disable Windows Event Logging
0.300000	T1070.003	Indicator Removal on Host	Clear Command History
0.300000	T1534	Internal Spearphishing	NaN
0.300000	T1553.003	Subvert Trust Controls	SIP and Trust Provider Hijacking
0.300000	T1574.002	Hijack Execution Flow	DLL Side-Loading
0.300000	T1055.008	Process Injection	Ptrace System Calls
0.300000	T1059.002	Command and Scripting Interpreter	AppleScript
0.300000	T1098.002	Account Manipulation	Exchange Email Delegate Permissions
0.300000	T1550.003	Use Alternate Authentication Material	Pass the Ticket
0.300000	T1556.003	Modify Authentication Process	Pluggable Authentication Modules
0.300000	T1559.002	Inter-Process Communication	Dynamic Data Exchange
0.300000	T1546.008	Event Triggered Execution	Accessibility Features
0.300000	T1548.004	Abuse Elevation Control Mechanism	Elevated Execution with Prompt

0.297292	T1001	Data Obfuscation	NaN
0.295703	T1033	System Owner/User Discovery	NaN
0.295238	T1218.008	Signed Binary Proxy Execution	Odbcconf
0.290476	T1546.013	Event Triggered Execution	PowerShell Profile
0.285714	T1020.001	Automated Exfiltration	Traffic Duplication
0.285714	T1110.002	Brute Force	Password Cracking
0.285714	T1578.002	Modify Cloud Compute Infrastructure	Create Cloud Instance
0.285714	T1578.003	Modify Cloud Compute Infrastructure	Delete Cloud Instance
0.285714	T1071.002	Application Layer Protocol	File Transfer Protocols
0.285714	T1071.003	Application Layer Protocol	Mail Protocols
0.285714	T1134.001	Access Token Manipulation	Token Impersonation/Theft
0.285714	T1554	Compromise Client Software Binary	NaN
0.280952	T1114.002	Email Collection	Remote Email Collection
0.280952	T1087.001	Account Discovery	Local Account
0.280952	T1546.006	Event Triggered Execution	LC_LOAD_DYLIB Addition
0.280952	T1547.007	Boot or Logon Autostart Execution	Re-opened Applications
0.280952	T1566.003	Phishing	Spearphishing via Service
0.280952	T1562.007	Impair Defenses	Disable or Modify Cloud Firewall
0.280952	T1578	Modify Cloud Compute Infrastructure	NaN
0.280952	T1578.001	Modify Cloud Compute Infrastructure	Create Snapshot
0.276190	T1053.006	Scheduled Task/Job	Systemd Timers
0.276190	T1218.002	Signed Binary Proxy Execution	Control Panel
0.276190	T1546.002	Event Triggered Execution	Screensaver
0.276190	T1134.002	Access Token Manipulation	Create Process with Token

0.271429	T1218.010	Signed Binary Proxy Execution	Regsvr32
0.271429	T1562.003	Impair Defenses	Impair Command History Logging
0.271429	T1574.012	Hijack Execution Flow	COR_PROFILER
0.271429	T1558.001	Steal or Forge Kerberos Tickets	Golden Ticket
0.270532	T1202	Indirect Command Execution	NaN
0.269764	T1069	Permission Groups Discovery	NaN
0.266667	T1547.003	Boot or Logon Autostart Execution	Time Providers
0.266667	T1574.004	Hijack Execution Flow	Dylib Hijacking
0.266667	T1127.001	Trusted Developer Utilities Proxy Execution	MSBuild
0.262859	T1200	Hardware Additions	NaN
0.261905	T1070.008	Indicator Removal	Clear Mailbox Data
0.261905	T1137.001	Office Application Startup	Office Template Macros
0.261905	T1546.004	Event Triggered Execution	.bash_profile and .bashrc
0.261905	T1547.009	Boot or Logon Autostart Execution	Shortcut Modification
0.260630	T1057	Process Discovery	NaN
0.260519	T1132	Data Encoding	NaN
0.258423	T1216	Signed Script Proxy Execution	NaN
0.257143	T1566.002	Phishing	Spearphishing Link
0.257143	T1087.004	Account Discovery	Cloud Account
0.257143	T1069.002	Permission Groups Discovery	Domain Groups
0.257143	T1543.001	Create or Modify System Process	Launch Agent
0.254762	T1568	Dynamic Resolution	NaN
0.252381	T1562.008	Impair Defenses	Disable Cloud Logs
0.252381	T1055.001	Process Injection	Dynamic-link Library Injection
0.252381	T1055.009	Process Injection	Proc Memory
0.250127	T1104	Multi-Stage Channels	NaN
0.247619	T1037.004	Boot or Logon Initialization Scripts	Rc.common

0.247619	T1134.005	Access Token Manipulation	SID-History Injection
0.247619	T1199	Trusted Relationship	NaN
0.247619	T1564.003	Hide Artifacts	Hidden Window
0.247619	T1114.003	Email Collection	Email Forwarding Rule
0.242857	T1025	Data from Removable Media	NaN
0.242857	T1092	Communication Through Removable Media	NaN
0.242857	T1134.003	Access Token Manipulation	Make and Impersonate Token
0.242857	T1559.001	Inter-Process Communication	Component Object Model
0.238095	T1137.003	Office Application Startup	Outlook Forms
0.238095	T1102.001	Web Service	Dead Drop Resolver
0.238095	T1550.004	Use Alternate Authentication Material	Web Session Cookie
0.238095	T1069.001	Permission Groups Discovery	Local Groups
0.235756	T1220	XSL Script Processing	NaN
0.233333	T1011.001	Exfiltration Over Other Network Medium	Exfiltration Over Bluetooth
0.233333	T1564.006	Hide Artifacts	Run Virtual Instance
0.233333	T1037.002	Boot or Logon Initialization Scripts	Logon Script (Mac)
0.233333	T1037.003	Boot or Logon Initialization Scripts	Network Logon Script
0.233333	T1070.004	Indicator Removal on Host	File Deletion
0.233333	T1137.004	Office Application Startup	Outlook Home Page
0.233333	T1137.005	Office Application Startup	Outlook Rules
0.233333	T1543.004	Create or Modify System Process	Launch Daemon
0.233333	T1546.010	Event Triggered Execution	AppInit DLLs
0.233333	T1053.004	Scheduled Task/Job	Launchd
0.233333	T1056.003	Input Capture	Web Portal Capture

0.233333	T1102.002	Web Service	Bidirectional Communication
0.233333	T1102.003	Web Service	One-Way Communication
0.229423	T1561	Disk Wipe	NaN
0.228571	T1070.007	Indicator Removal	Clear Network Connection History and Configurations
0.228571	T1137.002	Office Application Startup	Office Test
0.228571	T1546.014	Event Triggered Execution	Emond
0.228571	T1090.003	Proxy	Multi-hop Proxy
0.224270	T1573	Encrypted Channel	NaN
0.223810	T1205	Traffic Signaling	NaN
0.223810	T1195.003	Supply Chain Compromise	Compromise Hardware Supply Chain
0.223810	T1538	Cloud Service Dashboard	NaN
0.223810	T1573.001	Encrypted Channel	Symmetric Cryptography
0.219048	T1037.005	Boot or Logon Initialization Scripts	Startup Items
0.219048	T1546.011	Event Triggered Execution	Application Shimming
0.219048	T1606.002	Forge Web Credentials	SAML Tokens
0.214286	T1547.012	Boot or Logon Autostart Execution	Print Processors
0.214286	T1564.007	Hide Artifacts	VBA Stomping
0.214286	T1580	Cloud Infrastructure Discovery	NaN
0.210643	T1201	Password Policy Discovery	NaN
0.209524	T1553.004	Subvert Trust Controls	Install Root Certificate
0.209524	T1055.012	Process Injection	Process Hollowing
0.209524	T1090.001	Proxy	Internal Proxy
0.207413	T1135	Network Share Discovery	NaN
0.207143	T1561.001	Disk Wipe	Disk Content Wipe
0.205468	T1016	System Network Configuration Discovery	NaN
0.204762	T1560.001	Archive Collected Data	Archive via Utility
0.204762	T1564.004	Hide Artifacts	NTFS File Attributes
0.204762	T1205.001	Traffic Signaling	Port Knocking

0.204762	T1568.002	Dynamic Resolution	Domain Generation Algorithms
0.204762	T1001.003	Data Obfuscation	Protocol Impersonation
0.204762	T1008	Fallback Channels	NaN
0.202381	T1491	Defacement	NaN
0.202381	T1491.001	Defacement	Internal Defacement
0.200000	T1114.001	Email Collection	Local Email Collection
0.200000	T1552.003	Unsecured Credentials	Bash History
0.200000	T1090.002	Proxy	External Proxy
0.195238	T1606.001	Forge Web Credentials	Web Cookies
0.195238	T1547.008	Boot or Logon Autostart Execution	LSASS Driver
0.195238	T1567.001	Exfiltration Over Web Service	Exfiltration to Code Repository
0.195238	T1484.001	Domain Policy Modification	Group Policy Modification
0.195238	T1030	Data Transfer Size Limits	NaN
0.195238	T1055.002	Process Injection	Portable Executable Injection
0.195238	T1055.003	Process Injection	Thread Execution Hijacking
0.192857	T1491.002	Defacement	External Defacement
0.190476	T1027.002	Obfuscated Files or Information	Software Packing
0.190476	T1029	Scheduled Transfer	NaN
0.190476	T1056.002	Input Capture	GUI Input Capture
0.190476	T1132.001	Data Encoding	Standard Encoding
0.185714	T1055.013	Process Injection	Process Doppelg'e4nging
0.185714	T1001.001	Data Obfuscation	Junk Data
0.185714	T1001.002	Data Obfuscation	Steganography
0.185714	T1055.004	Process Injection	Asynchronous Procedure Call
0.185714	T1055.005	Process Injection	Thread Local Storage
0.185714	T1055.011	Process Injection	Extra Window Memory Injection
0.185714	T1055.014	Process Injection	VDSO Hijacking
0.185714	T1132.002	Data Encoding	Non-Standard Encoding
0.183333	T1565.002	Data Manipulation	Transmitted Data Manipulation
0.183333	T1598	Phishing for Information	NaN
0.183333	T1598.002	Phishing for Information	Spearphishing Attachment

0.180952	T1484.002	Domain Policy Modification	Domain Trust Modification
0.180952	T1547.002	Boot or Logon Autostart Execution	Authentication Package
0.180952	T1547.005	Boot or Logon Autostart Execution	Security Support Provider
0.180952	T1574.006	Hijack Execution Flow	LD_PRELOAD
0.178571	T1499	Endpoint Denial of Service	NaN
0.178571	T1499.004	Endpoint Denial of Service	Application or System Exploitation
0.176190	T1535	Unused/Unsupported Cloud Regions	NaN
0.176190	T1567.002	Exfiltration Over Web Service	Exfiltration to Cloud Storage
0.176190	T1553.001	Subvert Trust Controls	Gatekeeper Bypass
0.176190	T1555.005	Credentials from Password Stores	Password Managers
0.176190	T1049	System Network Connections Discovery	NaN
0.176190	T1216.001	Signed Script Proxy Execution	PubPrn
0.173810	T1498	Network Denial of Service	NaN
0.173810	T1585.003	Establish Accounts	Cloud Accounts
0.171725	T1129	Shared Modules	NaN
0.171429	T1137.006	Office Application Startup	Add-ins
0.171429	T1546.015	Event Triggered Execution	Component Object Model Hijacking
0.171429	T1036.007	Masquerading	Double File Extension
0.169048	T1499.001	Endpoint Denial of Service	OS Exhaustion Flood
0.166667	T1011	Exfiltration Over Other Network Medium	NaN
0.166667	T1053.007	Scheduled Task/Job	Container Orchestration Job
0.166667	T1564.002	Hide Artifacts	Hidden Users
0.166667	T1036.001	Masquerading	Invalid Code Signature
0.164286	T1499.002	Endpoint Denial of Service	Service Exhaustion Flood

0.164286	T1499.003	Endpoint Denial of Service	Application Exhaustion Flood
0.162681	T1518	Software Discovery	NaN
0.161905	T1555.002	Credentials from Password Stores	Securityd Memory
0.157354	T1020	Automated Exfiltration	NaN
0.157143	T1556.002	Modify Authentication Process	Password Filter DLL
0.157143	T1568.003	Dynamic Resolution	DNS Calculation
0.154762	T1598.003	Phishing for Information	Spearphishing Link
0.152381	T1037.001	Boot or Logon Initialization Scripts	Logon Script (Windows)
0.152381	T1546.009	Event Triggered Execution	AppCert DLLs
0.150000	T1498.002	Network Denial of Service	Reflection Amplification
0.147619	T1526	Cloud Service Discovery	NaN
0.145238	T1498.001	Network Denial of Service	Direct Network Flood
0.145238	T1598.001	Phishing for Information	Spearphishing Service
0.142857	T1552.007	Unsecured Credentials	Container API
0.138095	T1027.004	Obfuscated Files or Information	Compile After Delivery
0.138095	T1555.001	Credentials from Password Stores	Keychain
0.138095	T1555.003	Credentials from Password Stores	Credentials from Web Browsers
0.138095	T1574.011	Hijack Execution Flow	Services Registry Permissions Weakness
0.138095	T1115	Clipboard Data	NaN
0.133333	T1204.003	User Execution	Malicious Image
0.133333	T1518.001	Software Discovery	Security Software Discovery
0.133333	T1546.001	Event Triggered Execution	Change Default File Association
0.133333	T1547.010	Boot or Logon Autostart Execution	Port Monitors
0.133333	T1553.006	Subvert Trust Controls	Code Signing Policy Modification
0.133333	T1123	Audio Capture	NaN
0.133333	T1621	Multi-Factor Authentication Request Generation	NaN

0.128571	T1611	Escape to Host	NaN
0.128571	T1610	Deploy Container	NaN
0.128571	T1027.005	Obfuscated Files or Information	Indicator Removal from Tools
0.128571	T1070.006	Indicator Removal on Host	Timestomp
0.128571	T1568.001	Dynamic Resolution	Fast Flux DNS
0.126190	T1531	Account Access Removal	NaN
0.123810	T1070.005	Indicator Removal on Host	Network Share Connection Removal
0.123810	T1546.012	Event Triggered Execution	Image File Execution Options Injection
0.123810	T1556.007	Modify Authentication Process	Hybrid Identity
0.123810	T1007	System Service Discovery	NaN
0.123810	T1027.003	Obfuscated Files or Information	Steganography
0.119922	T1014	Rootkit	NaN
0.119048	T1039	Data from Network Shared Drive	NaN
0.119048	T1615	Group Policy Discovery	NaN
0.119048	T1074.001	Data Staged	Local Data Staging
0.119048	T1564.001	Hide Artifacts	Hidden Files and Directories
0.119048	T1124	System Time Discovery	NaN
0.119048	T1134.004	Access Token Manipulation	Parent PID Spoofing
0.118264	T1120	Peripheral Device Discovery	NaN
0.114286	T1036.004	Masquerading	Masquerade Task or Service
0.114286	T1217	Browser Bookmark Discovery	NaN
0.114286	T1218.014	Signed Binary Proxy Execution	MMC
0.114286	T1647	Plist File Modification	NaN
0.114286	T1027.001	Obfuscated Files or Information	Binary Padding
0.114286	T1218.013	Signed Binary Proxy Execution	Mavinject
0.114286	T1574.013	Hijack Execution Flow	KernelCallbackTable

0.109524	T1056.001	Input Capture	Keylogging
0.109524	T1006	Direct Volume Access	NaN
0.109524	T1546.007	Event Triggered Execution	Netsh Helper DLL
0.109524	T1547.013	Boot or Logon Autostart Execution	XDG Autostart Entries
0.109524	T1555.004	Credentials from Password Stores	Windows Credential Manager
0.109524	T1560.003	Archive Collected Data	Archive via Custom Method
0.109524	T1564.005	Hide Artifacts	Hidden File System
0.109524	T1010	Application Window Discovery	NaN
0.109524	T1036.006	Masquerading	Space after Filename
0.109524	T1090.004	Proxy	Domain Fronting
0.109524	T1553.002	Subvert Trust Controls	Code Signing
0.109524	T1578.004	Modify Cloud Compute Infrastructure	Revert Cloud Instance
0.109524	T1613	Container and Resource Disocovery	NaN
0.104762	T1069.003	Permission Groups Discovery	Cloud Groups
0.104762	T1070.009	Indicator Removal	Clear Persistence
0.104762	T1074.002	Data Staged	Remote Data Staging
0.104762	T1207	Rogue Domain Controller	NaN
0.104762	T1505.005	Server Software Component	Terminal Services DLL
0.104762	T1562.009	Impair Defenses	Safe Mode Boot
0.104762	T1614.001	System Location Discovery	System Language Discovery
0.104762	T1016.001	System Network Configuration Discovery	Internet Connection Discovery
0.104762	T1027.006	Obfuscated Files or Information	HTML Smuggling
0.104762	T1125	Video Capture	NaN
0.104762	T1497.001	Virtualization/Sandbox Evasion	System Checks
0.104762	T1497.003	Virtualization/Sandbox Evasion	Time Based Evasion
0.104762	T1609	Container Administration Command	NaN

0.104762	T1614	System Location Discovery	NaN
0.104762	T1620	Reflective Code Loading	NaN
0.100000	T1043	Commonly Used Port	NaN
0.100000	T1061	Graphical User Interface	NaN
0.100000	T1108	Redundant Access	NaN
0.100000	T1648	Serverless Execution	NaN
0.100000	T1026	Multiband Communication	NaN
0.100000	T1027.007	Obfuscated Files or Information	Dynamic API Resolution
0.100000	T1051	Shared Webroot	NaN
0.100000	T1064	Scripting	NaN
0.100000	T1149	LC_MAIN Hijacking	NaN
0.100000	T1205.002	Traffic Signaling	Socket Filters
0.100000	T1505.004	Server Software Component	IIS Components
0.100000	T1542.002	Pre-OS Boot	Component Firmware
0.100000	T1546.005	Event Triggered Execution	Trap
0.100000	T1546.016	Event Triggered Execution	Installer Packages
0.100000	T1547.014	Boot or Logon Autostart Execution	Active Setup
0.100000	T1556.005	Modify Authentication Process	Reversible Encryption
0.100000	T1560.002	Archive Collected Data	Archive via Library
0.100000	T1564.008	Hide Artifacts	Email Hiding Rules
0.100000	T1564.009	Hide Artifacts	Resource Forking
0.100000	T1612	Build Image on Host	NaN
0.100000	T1622	Debugger Evasion	NaN
0.100000	T1649	Steal or Forge Authentication Certificates	NaN
0.100000	T1027.008	Obfuscated Files or Information	Stripped Payloads
0.100000	T1027.009	Obfuscated Files or Information	Embedded Payloads
0.100000	T1034	Path Interception	NaN
0.100000	T1036.002	Masquerading	Right-to-Left Override
0.100000	T1055.015	Process Injection	ListPlanting
0.100000	T1056.004	Input Capture	Credential API Hooking
0.100000	T1062	Hypervisor	NaN

0.100000	T1087.003	Account Discovery	Email Account
0.100000	T1098.005	Account Manipulation	Device Registration
0.100000	T1153	Source	NaN
0.100000	T1213.003	Data from Information Repositories	Code Repositories
0.100000	T1480	Execution Guardrails	NaN
0.100000	T1480.001	Execution Guardrails	Environmental Keying
0.100000	T1497.002	Virtualization/Sandbox Evasion	User Activity Based Checks
0.100000	T1553.005	Subvert Trust Controls	Mark-of-the-Web Bypass
0.100000	T1556.006	Modify Authentication Process	Multi-Factor Authentication
0.100000	T1562.010	Impair Defenses	Downgrade Attack
0.100000	T1619	Cloud Storage Object Discovery	NaN
0.100000	T1175	Component Object Model and Distributed COM	NaN
0.097619	T1587	Develop Capabilities	NaN
0.092857	T1529	System Shutdown/Reboot	NaN
0.092857	T1588.001	Obtain Capabilities	Malware
0.078571	T1590.004	Gather Victim Network Information	Network Topology
0.078571	T1591.004	Gather Victim Org Information	Identify Roles
0.074106	T1496	Resource Hijacking	NaN
0.073810	T1595.001	Active Scanning	Scanning IP Blocks
0.073810	T1586.003	Compromise Accounts	Cloud Accounts
0.069048	T1595	Active Scanning	NaN
0.069048	T1595.002	Active Scanning	Vulnerability Scanning
0.069048	T1589.003	Gather Victim Identity Information	Employee Names
0.069048	T1590	Gather Victim Network Information	NaN
0.069048	T1590.001	Gather Victim Network Information	Domain Properties
0.069048	T1590.002	Gather Victim Network Information	DNS
0.069048	T1590.005	Gather Victim Network Information	IP Addresses

0.069048	T1590.006	Gather Victim Network Information	Network Security Appliances
0.064286	T1587.004	Develop Capabilities	Exploits
0.064286	T1588.006	Obtain Capabilities	Vulnerabilities
0.064286	T1592.003	Gather Victim Host Information	Firmware
0.059524	T1583.007	Acquire Infrastructure	Serverless
0.059524	T1587.002	Develop Capabilities	Code Signing Certificates
0.059524	T1588.003	Obtain Capabilities	Code Signing Certificates
0.054762	T1589	Gather Victim Identity Information	NaN
0.054762	T1594	Search Victim-Owned Websites	NaN
0.054762	T1588	Obtain Capabilities	NaN
0.054762	T1589.001	Gather Victim Identity Information	Credentials
0.054762	T1592	Gather Victim Host Information	NaN
0.054762	T1608	Stage Capabilities	NaN
0.054019	T1584	Compromise Infrastructure	NaN
0.050000	T1583	Acquire Infrastructure	NaN
0.050000	T1583.001	Acquire Infrastructure	Domains
0.050000	T1584.001	Compromise Infrastructure	Domains
0.050000	T1584.002	Compromise Infrastructure	DNS Server
0.050000	T1585	Establish Accounts	NaN
0.050000	T1585.001	Establish Accounts	Social Media Accounts
0.050000	T1586	Compromise Accounts	NaN
0.050000	T1586.001	Compromise Accounts	Social Media Accounts
0.050000	T1589.002	Gather Victim Identity Information	Email Addresses
0.050000	T1600	Weaken Encryption	NaN
0.050000	T1600.001	Weaken Encryption	Reduce Key Space
0.050000	T1600.002	Weaken Encryption	Disable Crypto Hardware
0.050000	T1583.002	Acquire Infrastructure	DNS Server
0.050000	T1583.003	Acquire Infrastructure	Virtual Private Server
0.050000	T1583.004	Acquire Infrastructure	Server
0.050000	T1583.005	Acquire Infrastructure	Botnet
0.050000	T1583.006	Acquire Infrastructure	Web Services

0.050000	T1584.003	Compromise Infrastructure	Virtual Private Server
0.050000	T1584.004	Compromise Infrastructure	Server
0.050000	T1584.005	Compromise Infrastructure	Botnet
0.050000	T1584.006	Compromise Infrastructure	Web Services
0.050000	T1584.007	Compromise Infrastructure	Serverless
0.050000	T1585.002	Establish Accounts	Email Accounts
0.050000	T1586.002	Compromise Accounts	Email Accounts
0.050000	T1587.001	Develop Capabilities	Malware
0.050000	T1587.003	Develop Capabilities	Digital Certificates
0.050000	T1588.002	Obtain Capabilities	Tool
0.050000	T1588.004	Obtain Capabilities	Digital Certificates
0.050000	T1588.005	Obtain Capabilities	Exploits
0.050000	T1590.003	Gather Victim Network Information	Network Trust Dependencies
0.050000	T1591	Gather Victim Org Information	NaN
0.050000	T1591.001	Gather Victim Org Information	Determine Physical Locations
0.050000	T1591.002	Gather Victim Org Information	Business Relationships
0.050000	T1591.003	Gather Victim Org Information	Identify Business Tempo
0.050000	T1592.001	Gather Victim Host Information	Hardware
0.050000	T1592.002	Gather Victim Host Information	Software
0.050000	T1592.004	Gather Victim Host Information	Client Configurations
0.050000	T1593	Search Open Websites/Domains	NaN
0.050000	T1593.001	Search Open Websites/Domains	Social Media
0.050000	T1593.002	Search Open Websites/Domains	Search Engines
0.050000	T1593.003	Search Open Websites/Domains	Code Repositories

0.050000	T1596	Search Open Technical Databases	NaN
0.050000	T1596.001	Search Open Technical Databases	DNS/Passive DNS
0.050000	T1596.002	Search Open Technical Databases	WHOIS
0.050000	T1596.003	Search Open Technical Databases	Digital Certificates
0.050000	T1596.004	Search Open Technical Databases	CDNs
0.050000	T1596.005	Search Open Technical Databases	Scan Databases
0.050000	T1597	Search Closed Sources	NaN
0.050000	T1597.001	Search Closed Sources	Threat Intel Vendors
0.050000	T1597.002	Search Closed Sources	Purchase Technical Data
0.050000	T1608.001	Stage Capabilities	Upload Malware
0.050000	T1608.002	Stage Capabilities	Upload Tool
0.050000	T1608.003	Stage Capabilities	Install Digital Certificate
0.050000	T1608.004	Stage Capabilities	Drive-by Target
0.050000	T1608.005	Stage Capabilities	Link Target
0.050000	T1608.006	Stage Capabilities	SEO Poisoning

NTNU

Norwegian University of
Science and Technology