Mansoura University

Faculty of Computer & information Science

Computer Science Department

Artificial Intelligence

Malware Detection System

Supervised By

Prof. Ahmed Tolba

Prof. Samir Elmougy

Dr. Zahraa Tarek

2020/2021

A project submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science

Graduation Project

by

Muhammed Talaat

Youssef Hwidi

Mohamed Hasaballa

Mohamed Abdellatif

Mahmoud Srour

Aly Emad

Mohamed Algarah

Aya Adel

Amira Alaa

Amira Mohamed

Abstract

Many public and private institutions suffer from cyber-crimes. In recent times these crimes have increased including theft, encryption, destruction and sometimes removal of data. This data varies between text files, photos, videos, and confidential information for countries from geographical locations to weapons and dangerous political information using a dangerous weapon which is malware (malicious software).

We aim in this project to prevent the repeated attacks of these crimes and try to put an end to this phenomenon by using two weapons which are Machine Learning and Malware Analysis. Our project is based on a combination of two fields which are Cyber Security “Malware Analysis & Reverse Engineering” and Machine Learning.

Traditional malware detection engines rely on the use of signatures - unique values - and behavior analysis that have been manually selected by a malware researcher to identify the presence of malicious code while making sure there are no collisions in the non-malicious samples group. The problems with this approach are several among others. It’s usually easy to bypass (depending on the type of signature, the change of a single bit or just a few bytes in the malicious code could make the malware undetectable) and it doesn’t scale very well when the number of researchers is orders of magnitude smaller than the number of unique malware families they need to manually reverse engineer, identify and write signatures for.

Mainly, our goal is teaching a computer, more specifically an artificial neural network, to detect Windows malware without relying on any explicit

signatures database or behavior analysis that we'd need to create but by simply ingesting the dataset of malicious files we want to be able to detect and learn from it to distinguish between malicious code or not both inside the dataset itself but most importantly while processing new unseen samples.

We use …… tools and languages and apis …

Acknowledgment

Thanks to everyone who helped us to carry out our project.

We would like to express our deep gratitude and respect to **Prof. Ahmed Tolba, Prof. Samir Elmougy**, **and Dr. El-Zahraa Tarek,** our supervisors, for their professional guidance and valuable support, encouragement and useful criticism of this project.

We would also like to thank **Eng. Abdelrhman Gamal** for his advices and constructive suggestions during the planning and development of this project.

Finally, our thanks go to **our families** for their love, prayers, caring, sacrifices and preparing us for the future.

Table of Contents

Chapter 1

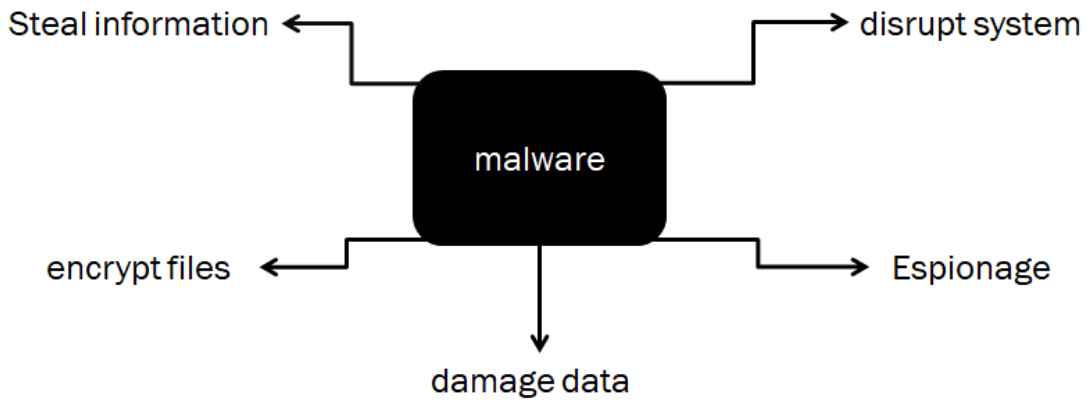
Introduction

Chapter 1

Introduction

1.1 Overview

Malware, short for malicious software, consists of programming (code, scripts, and other content) designed to disrupt operation or gather information that leads to loss of privacy, gain unauthorized access to system resources and other abusive behavior. Figure 1.1 shows the main threats of malware. It is a general term used to define a variety of forms of hostile, intrusive or annoying software or program code. Any software is classified as malware based on the intent of the maker rather than any particular feature. Malware includes computer viruses, worms, Trojan horses, spyware, dishonest adware, crime-ware, most rootkits and other malicious and unwanted software or program.



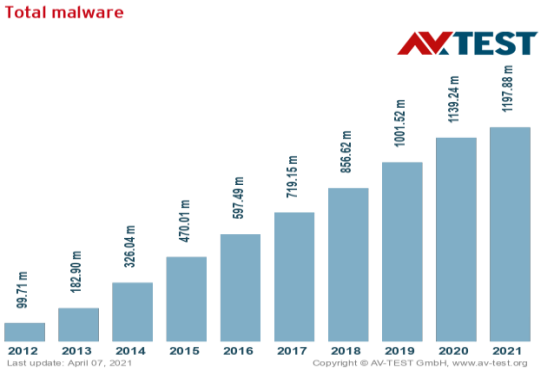
1.1.1 Motivation

Symantec published a report in 2008 that "the release rate of malicious code and other unwanted programs may be exceeding that of legitimate software applications." According to F-Secure, "As much malware produced in 2007 as in the previous 20 years altogether". While these may mean nothing to the average home user, these statistics are alarming keeping in mind the financial implications of such threats to enterprises in case such threats penetrate and compromise the large volumes of data stored and transacted upon.

Since the rise of widespread Internet access, malicious software has been designed for a profit, for example, forced advertising. Since 2003 the majority of viruses and worms have been designed to take control of users' computers for black market exploitation.

Spyware are programs designed to monitor users' web browsing and steal private information. Spyware programs do not spread like viruses but are installed by exploiting security holes or are packaged with user software.

Clearly, there is a very urgent need to find, not just a suitable method to detect infected files, but to build a smart engine that can detect new viruses by studying the structure of system calls made by malware.



Over 2 million Malware Per month.

Figure 1.1.1 Total malware [2]

Any corporation, company, bank or even important user needs malware detection system to detect any malicious file.

Malwares could steal important information or damage important data or freeze entire system or encrypt data and ask for ransom or make attackers control entire system.

We could see ransomware that encrypted a lot of corporations data and asked them for money, and Stuxnet worm the most dangerous malware that is a target supervisory control and data acquisition systems and that's believed to be responsible for causing substantial damage to the nuclear program of Iran.

There is a survey that indicates if it was measured as a country, then cybercrime caused more than 6 trillion dollars loss in 2021 and could be 10.5 trillion dollars by 2025. Figure 1.1.1 Total malware show it.

So, we decided to develop smart malware detection system to detect malwares with the help of machine learning to detect any malware and prevent it from passing our security system.

1.2 Project Objectives

- We aim to build an efficient system that prevents malwares to attack any other system.

- We are going to build that system using the combination of two fields, Cybersecurity and Artificial Intelligence.

- Detect known or unknown malicious files by the way they act and perform on the whole computers on system of the organization.

1.3 Project Scope

We have proposed a malware detection module based on advanced data mining and machine learning. While such a method may not be suitable for home users, being very processor heavy, this can be implemented at enterprise gateway level to act as a central antivirus engine to supplement anti-viruses present on end user computers. This will not only easily detect known viruses but act as a knowledge that will detect newer forms of harmful files. While a costly model requiring costly infrastructure, it can help in protecting invaluable enterprise data from security threat and prevent immense financial damage.

1.4 Project Contributions

Our proposed module based on advanced specific sequence of APIs into Memory from APTs attacks and machine learning to detect malicious activities like Process Hollowing, Process Doppelgänging, Reflective PE Injection, Thread Execution Hijacking, Dll injection, Anti-Debugger, Anti-VM, API Hooking, HTTP C&C Traffic APIs, Network Traffic Monitor APIs, Ransomware Encryption Algorithms APIs, KEYSTROKES Loggers APIs. This module can be implemented at enterprise gateway level to act as a central antivirus engine to supplement antiviruses present on end user computers. This will not only easily detect known viruses, but act as a knowledge that will detect newer forms of harmful files. While a costly model requiring costly infrastructure, it can help in protecting invaluable enterprise data from security threat, and prevent immense financial damage.

1.5 Project Organization

This project consists of six chapters. These chapters are organized to reflect the scientific steps toward our main objective.
A brief description of the contents of each chapter is given in the following paragraphs:

Chapter 1: introduction, introduces the project objectives, the scope of the project, the contribution of this project and project timeline.

Chapter 2: provides the reader with an overview of the literature review, background & related work & relationship between the relevant work and our own work.

Chapter 3: this chapter has the following objectives: software requirements, Uml diagrams & tools and languages.

Chapter 4: this chapter has the following objectives: system architecture, development methodology, data set and algorithm.

Chapter 5: this chapter has the following objectives: implementation, analysis and detection methodology, application anatomy and sample application code.

Chapter 6: conclusion and future work.

Chapter 2

Literature Review

Chapter 2

Literature Review

2.1 Background

2.1.1 Malicious software “Malware”

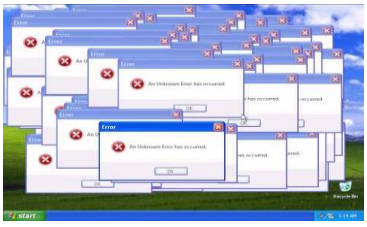
"Malware" is any malicious software or code that is harmful for systems. Malicious software can steal sensitive information from a system then uses it or slowing down the system gradually or even sending false emails from your email account without your knowledge.

2.1.2 Types of Malware

The six most common types of malicious software are: “Virus - Worms - Trojan - Ransomware - Spyware - Adware”

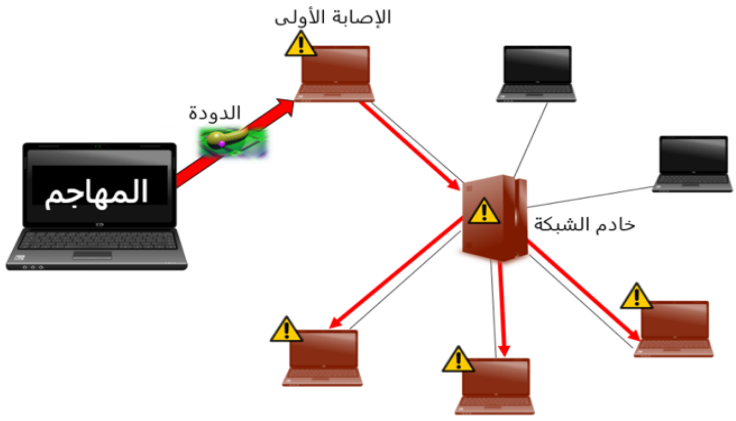
1 - Virus:

Viruses are designed to destroy the target computer or device by destroying the data, resetting the stationary disk or completely turning off your system. It can also be used to steal information, damage ers and networks, create robot comput networks, steal money, provide advertising and more. Computer viruses require a human procedure to infect computers and mobile devices and mail annexes and-are often spread through e Internet downloads.



2 - Worms:

n independent program that repeats itself to infect A worm is a other computers without having to take anaction from anyone but moves as soon as the device connects to computer networks by exploiting the weaknesses of the operating system. Because worms can pidly, they are often used to execute a series of code created spread ra to destroy the system to delete files on the device they reached (the host device) or to steal data from it and possibly encrypt them in preparation for ransom software attack.



3 - Trojan:

This type of malicious software enters your system undercover in a normal file image attached to an attractive email or ad that promises to make a lot of money without effort, but is actually designed to trick you into downloading malicious installing it on your software and device. Once a trojan horse is installed, you give cyber criminals access to your system, steal your

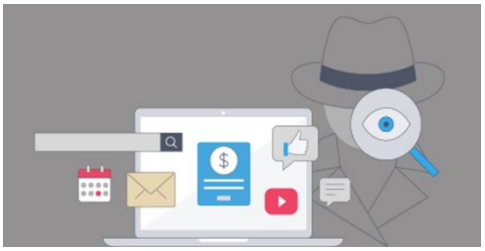
data, install more harmful software,they can modify files and maybe keyboard monitor all your activity on the device including recording all compressions like opening the microphone or camera without your knowledge and recording everything that happens in the device's perimeter.

4 - Ransomware

According to Cyberecurity Ventures, cyber crime is expected to cost the world 6 trillion dollars a year by 2021 as ransom programs generate a lot of money for cyber criminals. Ransom software is a type of malicious software that holds your data and asks you to pay for it to be recovered, i.e., it limits user access to the computer either by ypting files on the hard disk or locking the system and displaying encr messages on the screen aimed at forcing the user to pay the amount specified by the attacker in order to remove the malware and restore return to their access to the computer. Your system and data usually original state after being paid to the attacker, but it can't be guaranteed anyway, and we don't recommend it.

5 - Spyware

Spyware are installed on your device without your knowledge and are designed to track your browsing y on the Internet to see the activit sites you visit. Spyware can also include recording all keyboard compressors stealing login data to your postal or bank accounts, opening a microphone or camera without your knowledge, recording everything that urroundings of the devicehappens in the s, collecting account information, financial data and Spyware can reach your device by exploiting security gaps in your installed software or you can add spyware to another malicious program like trojan horses.



6 - Adware

The designers of these programs include ads with them in order to make money so that they show on the screen as soon as the program is installed. The purpose of these commercials may not necessarily be harmful; however, some sliding advertising programs are highly circumvented as they include promoting download of another software which means that they will open a door to malicious software from accessing your device. So, you have to take it away from your devices in order to avoid viruses, spy programs and other threats.

2.1.3 Malware detection methods

Malware is detected by two ways:

1 - Signature based detection:

Let's agree in the beginning that for every file in the world there is something called hash or signature MD5 and SHA 1 does not share it her file, and to simplify let's say that MDwith any ot5 or the SHA1 are like the personal identity of the files or the electronic fingerprint of the file which distinguishes each file.

When the attacker creates avirus, it will have a value or a hash or signature that does not exist for any other file or program. an MD5 hash In its lists of viruses (malware database), the process of adding viruses in databases is to add MD5 and SH A1 values to the database and not to add the virus itself. Iitself to the databases, we f it were to add viruses by smallest protection program with a siz would have thee of more than 100GB but only adds the MD 5 and SHA 1 hashes to its database.

Since every security program has a huge database of malware, rotection program takes the hash of when the user opens any file, the p the file and compares it with the hashes it has in its database. If the hash of the file matches a hash in the database, it will be considered a malicious file. Thus, it will delete it or inform the user that it is a icious program, whether he wants to delete it or not, and if it does mal not exist, it will be considered clean.

Is it all over here ? definitely not this was from the past. If ,it was just this, the protection programs would be able to detect all malicious ograms in record time and the story of the malware would end, but pr

the black hat owners certainly did not like this which led them to develop some new methods and techniques that enabled them to change the MD5 hash to another new hash ,unknown, falsified or blocked, as they developed many ways to bypass this type of malware detection.

Consequently, security companies have developed protection programs and developed their methods for examining files, and the sufficient to evaluate based detection technology is no longer-Signature the file as being healthy, but rather adopt other techniques and criteria to evaluate whether it is a malicious program or not.

2 - Anomaly based detection: ... "that we use"

ctivity It is based on anomalies statistics, where normal network a is determined in terms of several things, including:

- What are the protocols usually used?

- What ports and devices are generally connected to each other?

- It then alerts the administrator or user when an anomaly "abnormal" movement is detected.

How Anomaly based detection work:

In the Anomaly, malware is detected by ML, the system is trained on a specific data set that helps to form a baseline through which the normal and proper shape of the samples in the network is determined so that if there is an unusual sample "compared to baseline", it tries to enter the system after the training period is identified as malware, and the administrator is alerted.

Advantage of Anomaly-based:

It can detect many new malware whose signature is not yet known.

Disadvantage of Anomaly-based:

- Its accuracy is lower compared to signature.

- More false positive: “where it determines a specific sample as malware, but it is not”.

- It is very time consuming compared to signature.

- Uses a lot of memory compared to signature.

2.2 Review of Relevant work

2.2.1 Deep Neural Network based malware detection using two-dimensional binary program feature

In this paper we’ll be talking about deep neural network malware classifier that achieves a usable detection rate at an extremely low false positive rate and scales to real world training example volumes on commodity hardware. Specifically, we show that our system achieves a 95% detection rate at 0.1% false positive rate (FPR) based on more than 400,000 software binaries sourced directly from our customers and internal malware databases. We achieve these results by directly learning on all binaries without any filtering, unpacking or manually separating binary files into categories. Furthermore, we confirm our

false positive rates directly on a live stream of files coming in from Invincea's deployed endpoint solution providing an estimate of how many new binary files we expected to see a day on an enterprise network, describe how that relates to the false positive rate and translates into an intuitive threat score. Our results demonstrate that it is now feasible to quickly train and deploy a low resource, highly accurate machine learning classification model with false positive rates that approach traditional labor intensive signature based methods while also detecting previously unseen malware.

2.2.2 Flow-based malware detection using convolutional neural network

we suggest an automated malware detection method using convolutional neural network (CNN) and other machine learning algorithms. Lately malware detection methods have been dependent on the selected packet field of applications such as the port number and protocols, which is why those methods are vulnerable to malwares with unpredictable port numbers and protocols. The proposed method provides more robust and accurate malware detection, since it uses 35 different features extracted from packet flow, instead of the port numbers and protocols. Stratosphere IPS project data were used for evaluation, in which nine different public malware packets and normal state packets in an uninfected environment were converted to flow data with Netmate, and the 35-features were extracted from the flow data. CNN, multi-layer perceptron (MLP), support vector machine (SVM), and random forest (RF) were applied for classification, which showed >85% accuracy, precision and recall for all classes using CNN and RF.

2.2.3 REMOTE: robust external malware detection framework by using electromagnetic signals

Cyber-physical systems (CPS) are controlling many critical and sensitive aspects of our physical world while being continuously exposed to potential cyber-attacks. These systems typically have limited performance, memory, and energy reserves, which limits their ability to run existing advanced malware protection, and that, in turn, makes securing them very challenging. To tackle these problems, this paper proposes, REMOTE, a new robust framework to detect malware by externally observing Electromagnetic (EM) signals emitted by an electronic computing device (e.g., a microprocessor) while running a known application, in real-time and with a low detection latency, and without any a priori knowledge of the malware. REMOTE does not require any resources or infrastructure on, or any modifications to, the monitored system itself, which makes REMOTE especially suitable for malware detection on resource-constrained devices such as embedded devices, CPSs, and Internet of Things (IoT) devices where hardware and energy resources may be limited. To demonstrate the usability of REMOTE in real-world scenarios, we port two real-world programs (an embedded medical device and an industrial PID controller), each with a meaningful attack (a code-reuse and a code-injection attack), to four different hardware platforms. We also port shellcode-based DDoS and Ransomware attacks to five different standard applications on an embedded system. To further demonstrate the applicability of REMOTE to commercial CPS, we use REMOTE to monitor a Robotic Arm. Our

results on all these different hardware platforms show that, for all attacks on each of the platforms, REMOTE successfully detects each instance of an attack and has <; 0.1 percent false positives. We also systematically evaluate the robustness of REMOTE to interrupts and other system activity, to signal variation among different physical instances of the same device design, to changes overtime, and to plastic enclosures and nearby electronic devices. This evaluation includes hundreds of measurements and shows that REMOTE achieves excellent accuracy (<; 0.1 percent false positive and > 99.9 percent true positive rates) under all these conditions. We also compare REMOTE to prior work EDDIE and SYNDROME, and demonstrate that these prior work are unable to achieve high accuracy under these variations.

2.2.4 Project STAMINA Uses Deep Learning for Innovative Malware Detection

You're familiar with the phrase, "A picture is worth 1,000 words." Well, Microsoft and Intel are applying this philosophy to malware detection—using deep learning and a neural network to turn malware into images for analysis at scale. Project STAMINA—an acronym for Static Malware-as Image Network Analysis—converts malware samples into two-dimensional gray scale images that can be analyzed based on their unique crate There are three essential steps or stages of STAMINA: pre-processing, transfer learning, and evaluation.

Pre-processing takes care of image conversion. It directly converts the raw binary into its associated two-dimensional image.

Transfer learning is then performed on the resulting malware images as well as benign images. STAMINA transfer learning analysis was done using a Microsoft dataset of 2.2 million hashes of malware binaries. Using transfer learning enables STAMINA to accelerate training time and maintain high classification performance for the neural network.

The final step of STAMINA is evaluation. Evaluation considers accuracy, precision, recall, false positive rate, and other factors to produce a final result that achieved 99.07% accuracy with only a 2.58% false positive rate.

It is still in the semi early stages of development but results for Project STAMINA are promising so far. It is accurate and fast with smaller files, but researchers noted that it struggles against larger files (this is not a fundamental limit, but issues due to tool limitations). Once refined, though, Project STAMINA could be used as part of operating systems and anti-malware tools to proactively detect malware attacks.

2.3 Relationship between the Relevant work and our work

This system is similar to other systems in some things, for example:

1. The goal of the system is to detect the presence of malware on a host system.
2. Distinguishing whether a specific program is malicious or benign.

These properties that differentiate this system from others:

- Malware authors can write malware that could bypass any malware detection system easily, but we will use Machine Learning to make the computer learn and detect any new malware easily with high accuracy and we will achieve this by analysis a lot of malware samples with high level techniques and define a lot of malware characteristics, behaviors and use the best model to achieve this.
- When a new file has been accessed, our model will scan the file using the set of features of the most popular malicious file, this feature is a set of APIs that malicious files use it.

2.4 Summary

This system provides a solution to a problem which is detecting Windows Malware.

In this chapter we presented the systems that exist physically and offer the same service, but it is full of faults like:

- Unable to detect any malware.
- Easy to bypass.
- Require update database frequently.
- Rely on human expertise in creating the signature.

Therefore, these services that distinguish this system from others:

- The system is based on a combination of two fields
  Cyber Security “Malware Analysis” and Machine learning.
- System's goal is teaching a computer, more specifically an artificial neural network, to detect Windows malware without relying on

any explicit signatures database, that we'd need to create, but by simply ingesting the dataset of malicious files we want to be able to detect and learning from it to distinguish between malicious code or not.

- Detect known and unknown malware.

This chapter offers to the reader some relevant work to our project, we will show you some similar or related work with our application that will show the advantages and disadvantages of these applications and through which we were inspired by the idea of this application in order to bring together most of these features and solve most of these defects.

Chapter 3

Software Engineering Analysis and UML Diagrams

Chapter 3

Software Engineering

Analysis and UML Diagrams

3.1 Introduction

In this chapter we will provide the system analysis and design process that included in our system that includes the functional and non-functional requirements and UML Diagrams. There are 3 methods for developing a system:

1. Agile Development.
2. Waterfall Development.
3. Spiral Development.

In this project we will use Agile Development.

3.2 Agile Development Life Cycle

Agile development life cycle composed of getting requirements, design, development, testing, deployment and review.

3.3 Software requirements

✓ Software requirement divides into:

- **User requirements**
- **System requirements**

3.3.1 User Requirements

✓ There are two types of user requirements:

- **Functional Requirements**
- **Non-Functional Requirements**

3.3.1.1 Functional requirements

Functional Requirements: describes what a software system should do and describing the behavior of the system.

- ✓ Expect the entry of a virus in several cases such as:
  a) connecting to the Internet
  b) downloading files
  c) connecting to other strange devices on the network
  d) Open a new email address

- ✓ Permanent monitoring of the system and the data and their flow and operation.

- ✓ Monitor any change that occurs within the company's databases and other files such as:
  a) Transfer
  b) Encryption
  c) Destruction

- ✓ Permanent awareness of any change in the reading of internal systems such as the hardware for the system.

- ✓ Detect malicious files by the way they act and perform on the whole pcs on system of the organization.

3.3.1.2 Non-functional requirements

Non-Functional Requirements: place constraints on how the system will do the functional requirements and elaborates a performance characteristic of the system.

- ✠ High performance
- ✠ Accuracy
- ✠ Speed
- ✠ Availability all the time
- ✠ Portability
- ✠ Self-development ability
- ✠ Security
- ✠ Fault Tolerance
- ✠ Backup
- ✠ Maintainability

3.3.2 System Requirements

Function	Smart Recruitment Management System
Description	The system uses AI “Machine Learning” and Cyber security “Malware Analysis” to detect Windows malware without relying on any explicit signatures database.
Input	Any new file whether it is opened, downloaded or transferred.
Source	An internal access from an employee's device, Web sites, E-mails.
Output	This file is malicious or legitimate.
Action	- New file is accessed by any way. - The model will scan the file using the set of features of the most popular malicious file, this set feature is a set of APIs that malicious files use it, using one of the classification methods. - The model monitors the status of the organization system after activating or opening this new file if something is changed in the system, on its effect model decide it is malicious or legitimate.

System requirements are the configuration that a system must have in order for a hardware or software application to run smoothly and efficiently. Failure to meet these requirements can result in installation problems or performance problems. The former may prevent a device or application from getting installed, whereas the latter may cause a product to malfunction or perform below expectation or even to hang or crash.

3.4 UML Diagrams

UML stands for Unified Modeling Language, It is a methodology used in system analysis to identify, clarify, and organize system requirements, and how system component act with each other.

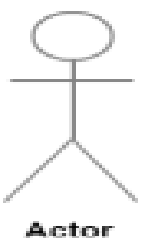
3.4.1 Use Case Diagrams

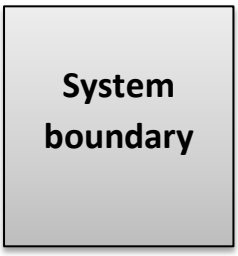
It is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved.

1) Components of Use Case

The elements that constitute a use case diagram:

- **Actor**: It is what interacts with a use case.
- **Use Case**: It is a visual representation of a distinct business functionality in a system.
- **System boundary**: It defines the scope of what a system will be.





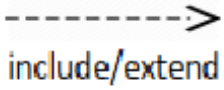
2) Relationships in Use Case

➢ **Include:**

When a use case is depicted as using the functionality of another use case in a diagram.

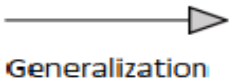
➢ **Extend:**

The child use case adds to the existing functionality and characteristics of the parent use case.



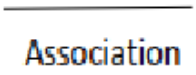
➢ **Generalizations:**

It is also a parent-child relationship between use cases.

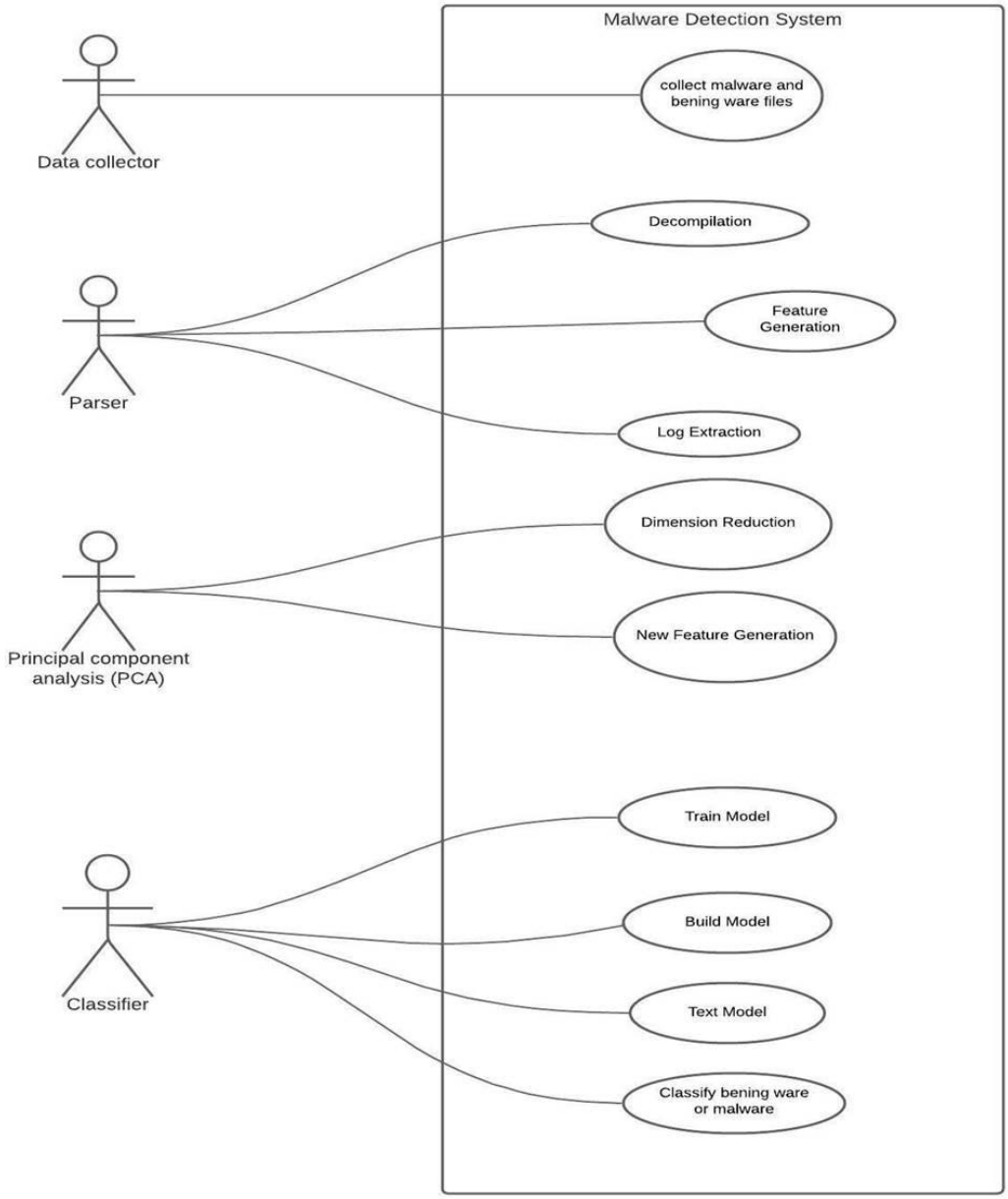


➢ **Association:**

Link between an actor and the use case.



- **Use Case Diagram of the system**



3.4.2 Sequence Diagrams

Sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

Sequence diagrams are sometimes called event diagrams or event scenarios.

1) Sequence Diagrams Notation

➢ **Actor:**

Represents a type of role where it interacts With the system and its objects.

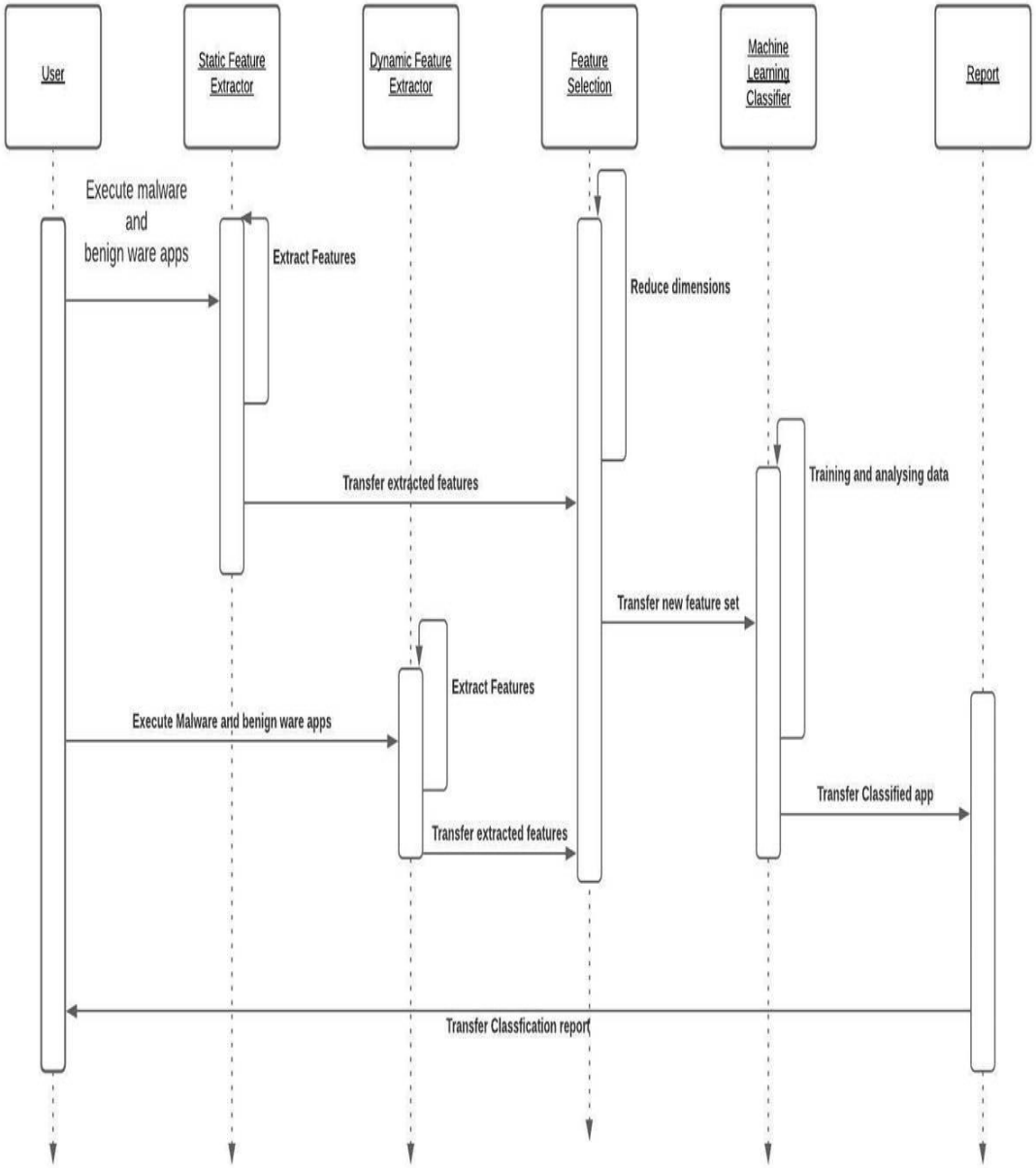
➢ **Lifelines:**

It's a named element which depicts an individual participant in a sequence diagram.

➢ **Messages:**

Communication between objects is Depicted using messages.

- **Sequence Diagrams of the system**

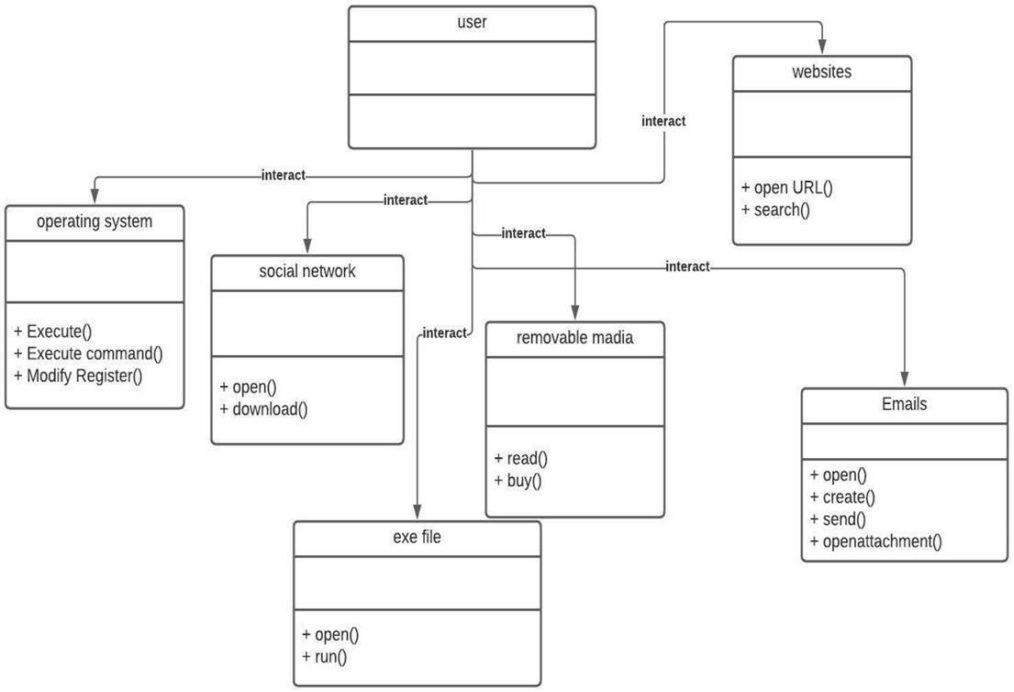


3.4.3 Class Diagram

It is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

The class diagram is the main building block of object-oriented modeling. It is used for general conceptual modeling of the structure of the application, and for detailed modeling translating the models into programming code. Class diagrams can also be used for data modeling.

- **Class Diagram of the system**

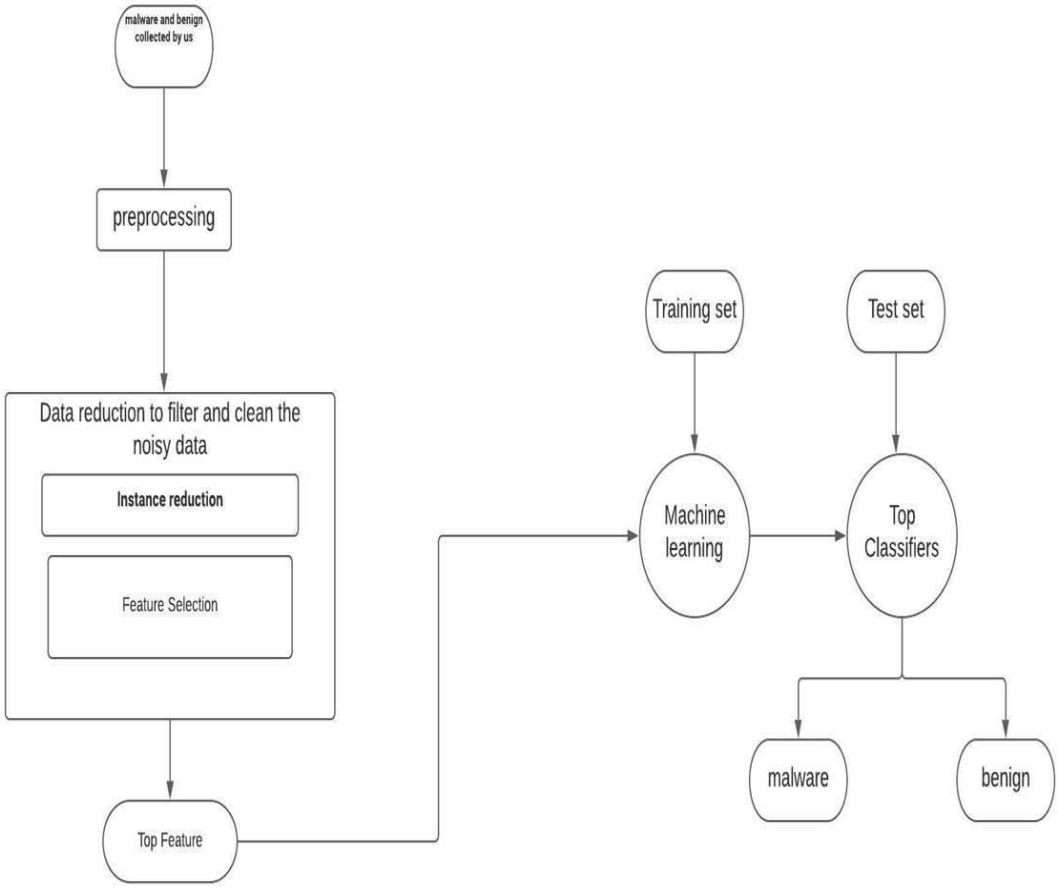


3.4.4 Activity Diagram

It illustrates the flow of control in a system and refers to the steps involved in the execution of a use case. It is a graphical representation of workflows of stepwise activities and actions with support for choice, iteration and concurrency.

In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e., workflows), as well as the data flows intersecting with the related activities.

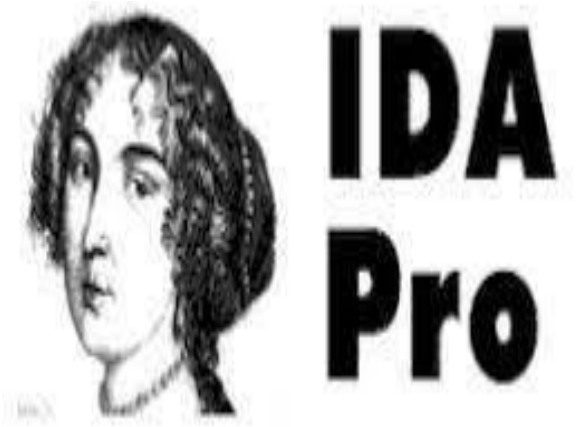
- **Activity Diagram of the system**



3.5 Tools and Languages

3.5.1 Tools

1) IDA Pro



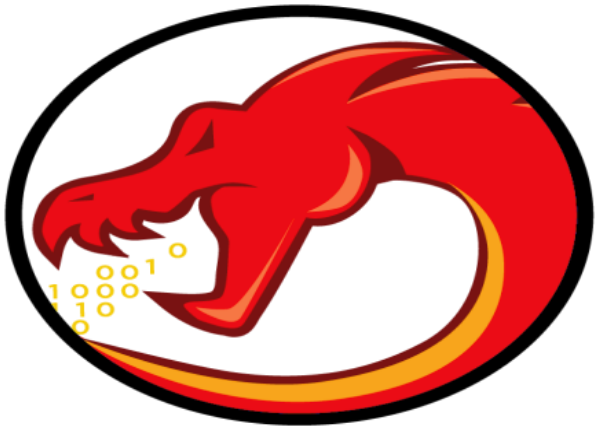
IDA stands for Interactive Disassembler; it is a disassembler for computer software which generates assembly language source code from machine-executable code. It supports a variety of executable formats for different processors and operating systems. It also can be used as a debugger for Windows PE, Mac OS X Mach-O, and Linux ELF executables.

A decompiler plug-in for programs compiled with a C/C++ compiler is available at extra cost. The latest full version of IDA Pro is commercial, while an earlier and less capable version is available for download free of charge.

IDA performs automatic code analysis, using cross-references between code sections, knowledge of parameters of API calls, and other information. However, the nature of disassembly precludes total accuracy, and a great deal of human intervention is necessarily

required; IDA has interactive functionality to aid in improving the disassembly. A typical IDA user will begin with an automatically generated disassembly listing and then convert sections from code to data and vice versa, rename, annotate, and otherwise add information to the listing, until it becomes clear what it does.

2) Ghidra



Ghidra is a free and open-source reverse engineering tool developed by the National Security Agency (NSA) of United States of America. The binaries were released at RSA Conference in March 2019; the sources were published one month later on GitHub, Ghidra is seen by many security researchers as a competitor to IDA Pro. The software is written in Java using the Swing framework for the GUI. The decompiler component is written in C++. Ghidra plugins can be developed in Java or in Python.

3) X64 dbg

x64dbg is an open-source debugger for Windows that is a popular malware analysis tool. A debugger is used to step through code as it executes, so you can see exactly what it's doing. Debuggers are essential for troubleshooting bugs, but they're also used to reverse engineer malware.

4) PE Studio

PeStudio is a portable tool that performs malware assessments on executable files, since the target file is never launched during the course of the investigation you can safely evaluate the file, in addition to malware, without risk.

The goal of **pestudio** is to spot artifacts of executable files in order to ease and accelerate Malware Initial Assessment. The tool is used

by Computer Emergency Response (CERT) teams, Security Operations Centers (SOC) and Labs worldwide.

5) PE-Bear

PE-bear is a reversing tool for PE files.

its Objective was to deliver fast and flexible "first view" tool for malware analysts.

Stable and capable to handle malformed PE files.

6) CFF Explorer

CFF Explorer was designed to make PE editing as easy as possible, but without losing sight on the portable executable's internal structure. This application includes a series of tools which might help not only reverse engineers but also programmers. It offers a multi-file environment and a switchable interface. Also, it's the first PE editor with full support for the .NET file format. With this tool you can easily edit metadata's fields and flags. If you're programming something

that has to do with .NET metadata, you will need this tool. The resource viewer supports .NET image formats like icons, bitmaps, pngs. You'll be able to analyze .NET files without having to install the .NET framework, this tool has its own functions to access the .NET format.

7) Visual Studio

Microsoft Visual Studio is an IDE made by Microsoft and used for different types of software development such as computer programs, websites, web apps, web services, and mobile apps. It contains completion tools, compilers, and other features to facilitate the software development process.

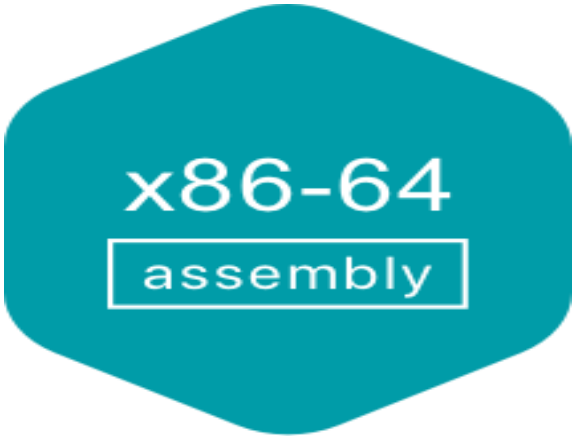
The **Visual Studio** IDE (integrated development environment) is a software program for developers to write and edit their code. Its user interface is used for software development to edit, debug and build code. Visual Studio includes a code editor supporting IntelliSense (the code completion component) as well as code refactoring. The integrated debugger works both as a source-level debugger and a machine-level debugger. Other built-in tools include a code profiler, designer for building GUI applications, web designer, class designer, and database schema designer.

8) Anaconda

Anaconda is a distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.), that aims to simplify package management and deployment. The distribution includes data-science packages suitable for Windows, Linux, and macOS. It is developed and maintained by Anaconda, Inc., which was founded by Peter Wang and Travis Oliphant in 2012. As an Anaconda, Inc. product, it is also known as **Anaconda Distribution** or **Anaconda Individual Edition**, while other products from the company are Anaconda Team Edition and Anaconda Enterprise Edition, both of which are not free.

3.5.2 Languages

1) x86 – x64 Assembly



Assembly language is a low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions.[2] Because assembly depends on the machine code instructions, every assembly language is designed for exactly one specific computer architecture. Assembly language may also be called symbolic machine code. **x86 assembly** language is a family of backward-compatible assembly languages, which provide some level of compatibility all the way back to the Intel 8008 introduced in April 1972. x86 assembly languages are used to produce object code for the x86 class of processors. x64 is a generic name for the 64-bit extensions to Intel's and AMD's 32-bit x86 instruction set architecture (ISA). AMD introduced the first version of x64, initially called x86-64 and later renamed AMD64.

2) C

C is a general-purpose, procedural computer programming language supporting structured programming, lexical variable scope, and recursion, with a static type system. By design, C provides constructs that map efficiently to typical machine instructions. It has found lasting use in applications previously coded in assembly language. Such applications include operating systems and various application software for computer architectures that range from supercomputers to PLCs and embedded systems, and it is an imperative procedural language. It was designed to be compiled to provide low-level access to memory and language constructs that map efficiently to machine instructions, all with minimal runtime support. Despite its low-level capabilities, the language was designed to encourage cross-platform programming. A standards-compliant C program written with portability in mind can be compiled for a wide variety of computer platforms and operating systems with few changes to its source code.

3) Python

Python is an interpreted high-level general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant indentation. Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. **Python** is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library. Guido van Rossum began working on Python in the late 1980s, as a successor to the ABC programming language, and first released it in 1991 as Python 0.9.0. Python 2.0 was released in 2000 and introduced new features, such as list comprehensions and a garbage collection system using reference counting. Python 3.0 was released in 2008 and was a major revision of the language that is not completely backward-compatible and much Python 2 code does not run

unmodified on Python 3. Python 2 was discontinued with version 2.7.18 in 2020.

Python consistently ranks as one of the most popular programming languages.

Chapter 4

System Design

Chapter 4

System Design

4.1 Overview

In this chapter we will cover the data set and algorithms that we use in our application.

4.2 Dataset

We collected two data sets, first data set we collect it from 2000 APTs Attacks, malware samples and another from Kaggle[31] by Angelo Oliveira.

4.2.1 Our Dataset

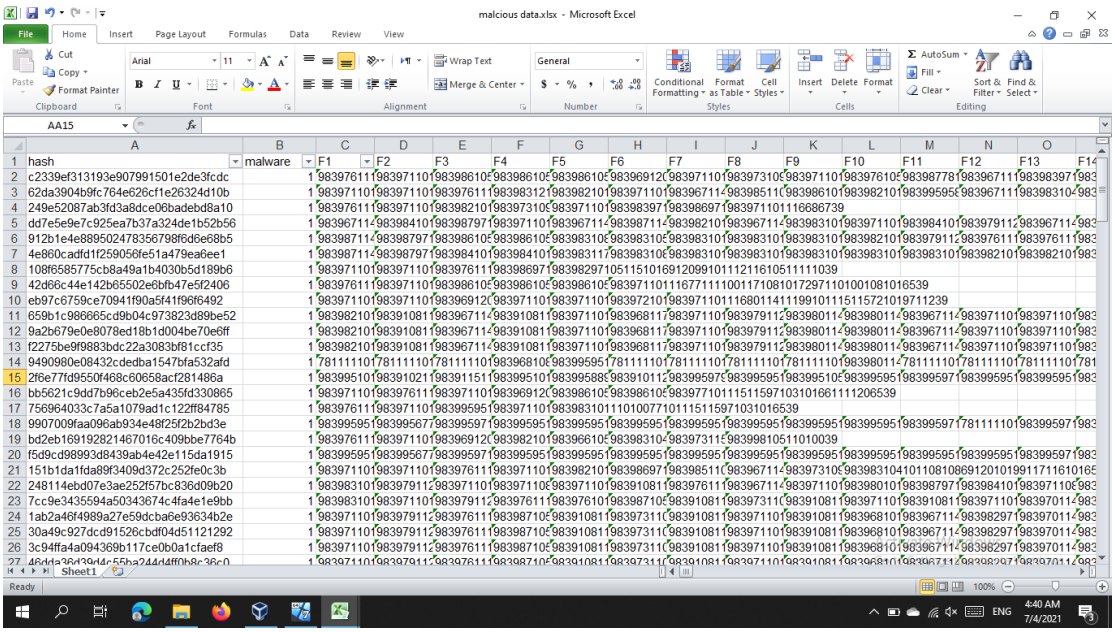
FEATURES:

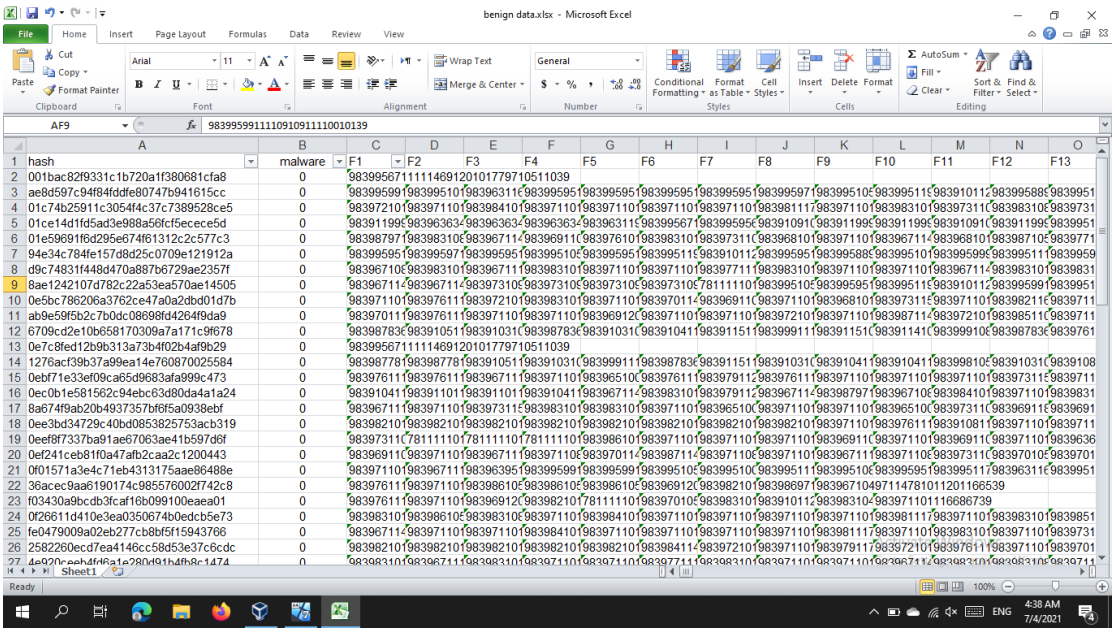
- **Column** name: hash
- Description: MD5 hash of the example
- Type: 32 bytes string

- **Column** name: malware
- Description: Class
- Type: Integer: 0 (Goodware) or 1 (Malware)
- **Column** name: F1 … F100

- Description: API call
- Type: ASCII code

we can see our Data-Set in the two figures below

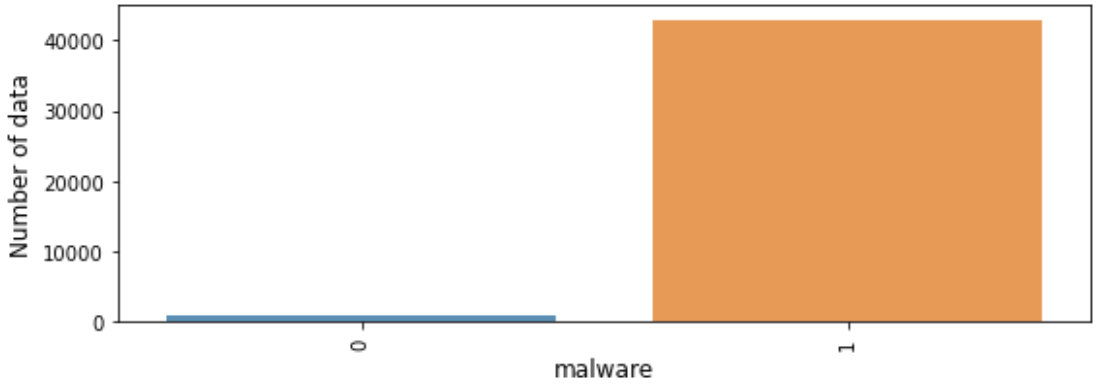




4.2.2 Dataset from Kaggle

It represents 43876 and contains 102 features.

	hash	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	...	t_91	t_92	t_93	t_94	t_95	t_96	t_97	t_98	t_99	malware
0	071e8c3f8922e186e57548cd4c703a5d	112	274	158	215	274	158	215	298	76	...	71	297	135	171	215	35	208	56	71	1
1	33f8e6d08a6aae939f25a8e0d63dd523	82	208	187	208	172	117	172	117	172	...	81	240	117	71	297	135	171	215	35	1
2	b68abd064e975e1c6d5f25e748663076	16	110	240	117	240	117	240	117	240	...	65	112	123	65	112	123	65	113	112	1
3	72049be7bd30ea61297ea624ae198067	82	208	187	208	172	117	172	117	172	...	208	302	208	302	187	208	302	228	302	1
4	c9b3700a77facf29172f32df6bc77f48	82	240	117	240	117	240	117	240	117	...	209	260	40	209	260	141	260	141	260	1



We do feature selection by used trees classifier (extra trees classifier()) to extract the most important feature and represent (31 feature) which leads to increasing speed and reducing time.

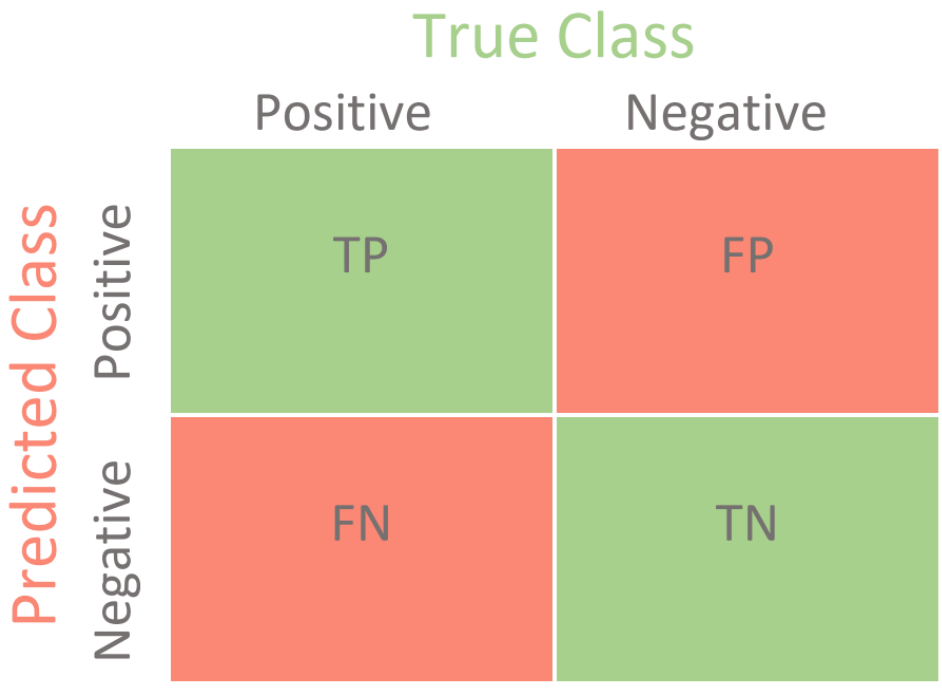
```
1. feature t_2 (0.044445)
2. feature t_77 (0.031929)
3. feature t_91 (0.020506)
4. feature t_1 (0.018732)
5. feature t_99 (0.017880)
6. feature t_80 (0.017809)
7. feature t_79 (0.017577)
8. feature t_90 (0.016558)
9. feature t_76 (0.016181)
10. feature t_3 (0.014531)
11. feature t_24 (0.013821)
12. feature t_94 (0.013649)
13. feature t_64 (0.013627)
14. feature t_21 (0.012499)
15. feature t_60 (0.012317)
16. feature t_47 (0.011796)
17. feature t_26 (0.011651)
18. feature t_69 (0.011573)
19. feature t_62 (0.011397)
20. feature t_58 (0.011220)
21. feature t_95 (0.011158)
22. feature t_53 (0.010894)
23. feature t_29 (0.010503)
24. feature t_81 (0.010468)
25. feature t_13 (0.010373)
26. feature t_84 (0.010324)
27. feature t_18 (0.010313)
28. feature t_88 (0.010261)
29. feature t_15 (0.010235)
30. feature t_82 (0.010115)
31. feature t_86 (0.010041)
```

We split the data set for training and testing data. 80% of the data used for training the machine learning algorithm. To ensure that the training of the classification algorithm can be well generalized to the new data. And 20% from the data used for testing the model.

4.3 The used Algorithms

We used the data set to test it from the algorithms and calculate the accuracy to find out which one is better. To calculate the accuracy we used confusion matrix.

Confusion matrix used in the field of machine learning and specifically the problem of statistical classification, a confusion matrix, also known as an error matrix is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix).



- Recall or true positive rate (TPR)

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{P}} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} = 1 - \mathrm{FNR}$$

- Selectivity or true negative rate (TNR)

$$\mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} = 1 - \mathrm{FPR}$$

- Miss rate or false negative rate (FNR)

$$\mathrm{FNR} = \frac{\mathrm{FN}}{\mathrm{P}} = \frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TP}} = 1 - \mathrm{TPR}$$

- Fall out or false positive rate (FPR)

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{N}} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} = 1 - \mathrm{TNR}$$

- Precision or positive predictive value (PPV)

$$\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} = 1 - \mathrm{FDR}$$

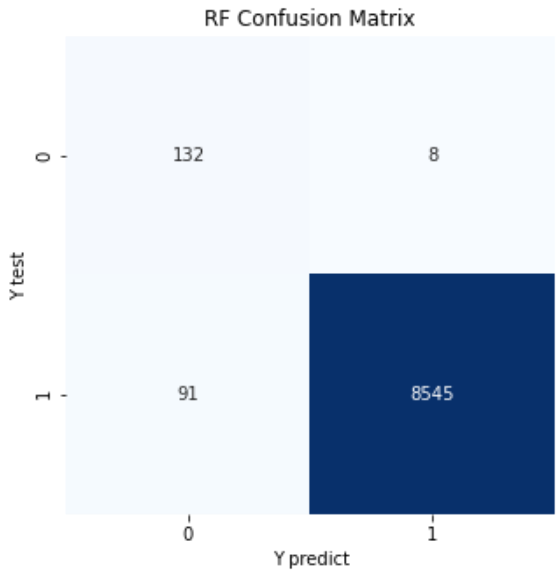
- Accuracy (ACC)

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{P} + \mathrm{N}} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

Some of algorithm that was trained and used to calculate accuracy:

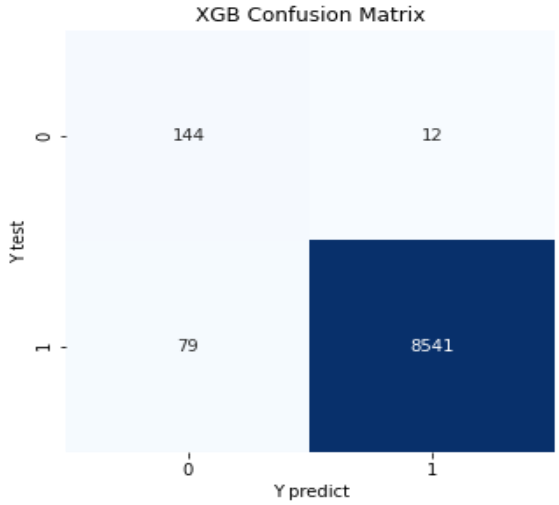
1- Random Forest Classifier

Accuracy of random forest classifier: 98.87%



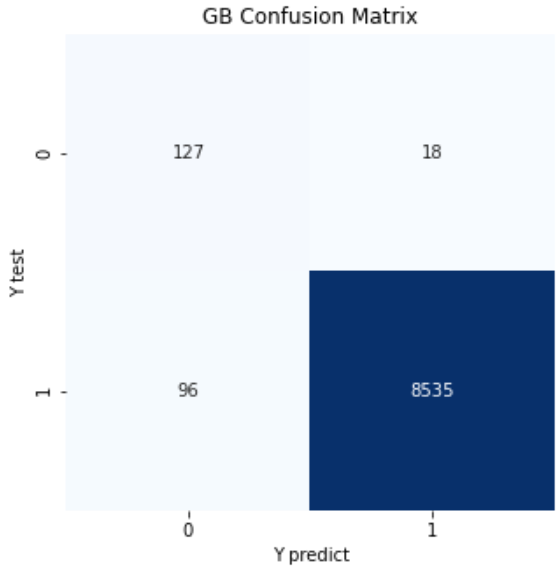
2- XGB Classifier

Accuracy of XGB classifier: 98.96%



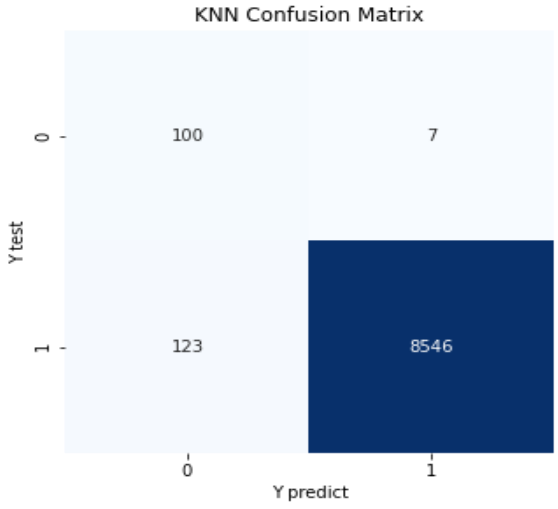
3- Gradient Boosting Classifier

Accuracy of gradient boosting classifier: 98.70%



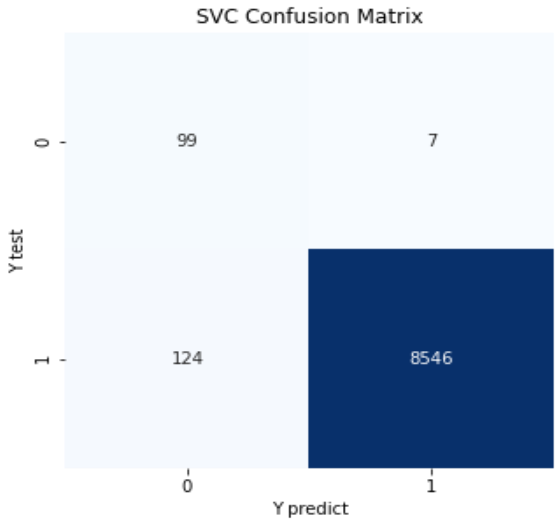
4- K-Nearest Neighbors (KNN)

Accuracy of K-nearest neighbors: 98.52%



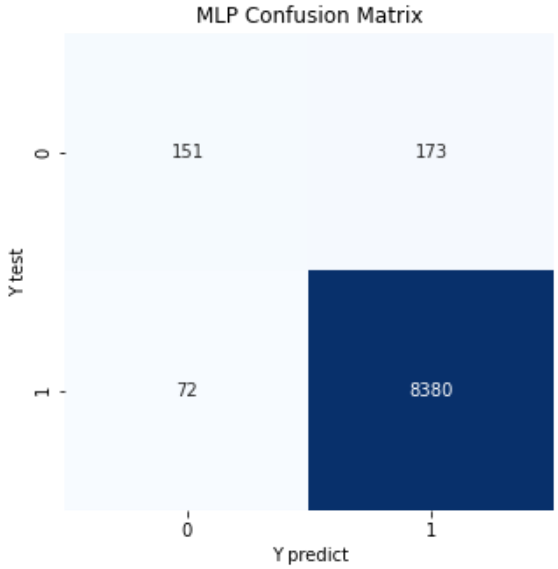
5- Support Vector Classifier (SVC)

Accuracy of Support vector classifier: 98.51%



6- Multilayer Perceptron (MLP) classifier

Accuracy of MLP classifier: 97.21%



Algorithm comparison:

RF: 98.871923 % // XGB: 98.963081 %

GB: 98.723792 % // KNN: 98.518687 %

SV: 98.507293 % // MLP: 97.208295 %

Winner algorithm is XGB with a 98.963081 %

success.

Classifier	TP	FN	FP	TN	Accuracy
RF	8545	91	8	133	98.87%
XGB	8541	79	12	144	98.96%
GB	8535	96	18	127	98.70%
KNN	8546	123	7	100	98.52%
SVC	8546	124	7	99	98.51%
MLP	8380	72	173	151	97.21%

4.4 Summary

In this chapter we discussed the system architecture, development methodology, data base design and algorithms.

Chapter 5

Implementation, Analysis and Detection Methodology

Chapter 5

Implementation, Analysis and Detection Methodology

In this chapter we'll be talking about how we implement our project, techniques we used and our methodology to detect new APTs Attacks malware samples.

5.1 Application Anatomy

5.1.1 Reverse Engineering and Malware analysis Phase

In our research we went to extract accurate features, we have relied in extracting these features that no malicious activities can occur on the device without the presence of one of them, and not only that, but it must be present based on a precise order that cannot be changed and thus this malicious software can be detected based on the presence of this information and its arrangement within memory, this features will be advanced specific sequence of APIs and malicious encrypted TLS traffic metadata from APTs attacks to detect malicious activities based

on a specific sequence of APIs techniques into the memory like Process Hollowing, Process Doppelgänging, Reflective PE Injection, Thread Execution Hijacking, DLL injection, Anti-Debugger, Anti-VM, API Hooking, malicious C&C traffic, HTTP C&C Traffic APIs, Network Traffic Monitor, Ransomware Encryption Algorithms APIs, KEYSTROKES Loggers APIs.

Now we will present some of our methodology, analysis and reverse engineering of advanced malware samples from APTs attacks.

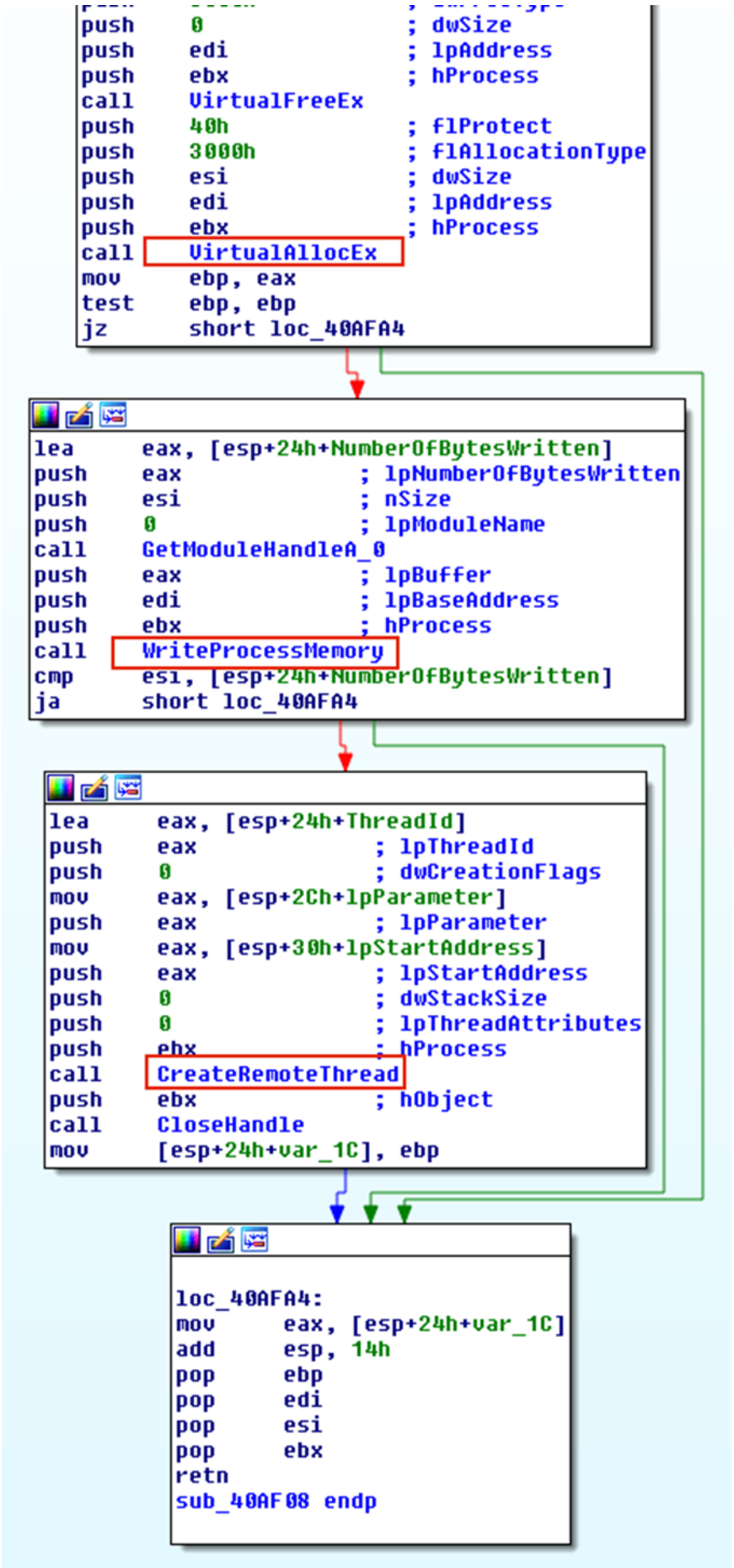
1- DLL injection using CreateRemoteThread():

This technique is one of the most common techniques used to inject malware into another process. The malware writes the path to its malicious dynamic-link library (DLL) in the virtual address space of another process, and ensures the remote process loads it by creating a remote thread in the target process, and based on our analysis and reverse engineering this is the specific sequence APIs any malware will use to implement this technique as we can see below :

- **OpenProcess():** enables the malicious process to talk to the targeted process.
- **VirtualAllocEx():** enables a process to allocate memory space in another process, malwares use it to store the string name of the malicious DLL.
- **WriteProcessMemory():** This enables a process to write into another process memory space. Due to this functionality, a victim process might be injected by a string that later may be loaded to **LoadLibraryA()** function calls.

- **GetModuleHandle():** Helps a process to define the way to access dlls that are loadedinto the memory space. Find the kernal32.dll (used to implement loadlibraryfunction)
- **GetProcAddress():** Can be used to find the LoadLibrary() address within thekernel32.dll file.
- **CreateRemoteThread():** Enables a malware to create a remote thread in a remote victim process. One of its arguments is the function address, which new thread will run.
- **LoadLibraryA** will run in the new created thread.
- **LocaLibraryA** Works on any application and used to load the malicious DLL to the remote address space.

The screenshot below displays the analysis and reverse engineering of a malware named **Rebhip** worm performing this technique with the APIs as we explained.


push 0 ; dwSize
push edi ; lpAddress
push ebx ; hProcess
call VirtualFreeEx
push 40h ; flProtect
push 3000h ; flAllocationType
push esi ; dwSize
push edi ; lpAddress
push ebx ; hProcess
call VirtualAllocEx
mov ebp, eax
test ebp, ebp
jz short loc_40AFA4
lea eax, [esp+24h+NumberOfBytesWritten]
push eax ; lpNumberOfBytesWritten
push esi ; nSize
push 0 ; lpModuleName
call GetModuleHandleA_0
push eax ; lpBuffer
push edi ; lpBaseAddress
push ebx ; hProcess
call WriteProcessMemory
cmp esi, [esp+24h+NumberOfBytesWritten]
ja short loc_40AFA4
lea eax, [esp+24h+ThreadId]
push eax ; lpThreadId
push 0 ; dwCreationFlags
mov eax, [esp+2Ch+lpParameter]
push eax ; lpParameter
mov eax, [esp+30h+lpStartAddress]
push eax ; lpStartAddress
push 0 ; dwStackSize
push 0 ; lpThreadAttributes
push ebx ; hProcess
call CreateRemoteThread
push ebx ; hObject
call CloseHandle
mov [esp+24h+var_1C], ebp
loc_40AFA4:
mov eax, [esp+24h+var_1C]
add esp, 14h
pop ebp
pop edi
pop esi
pop ebx
retn
sub_40AF08 endp

The malware first needs to target a process for injection (e.g. svchost.exe). This is usually done by searching through processes by calling a trio of Application Program Interfaces (APIs): **CreateToolhelp32Snapshot**, **Process32First**, and **Process32Next**. **CreateToolhelp32Snapshot** is an API used for enumerating heap or module states of a specified process or all processes, and it returns a snapshot. **Process32First** retrieves information about the first process in the snapshot, and then **Process32Next** is used in a loop to iterate through them. After finding the target process, the malware gets the handle of the target process by calling **OpenProcess**.

As shown in Figure below, the malware calls **VirtualAllocEx** to have a space to write the path to its DLL. The malware then calls **WriteProcessMemory** to write the path in the allocated memory. Finally, to have the code executed in another process, the malware calls APIs such as CreateRemoteThread, **NtCreateThreadEx**, or **RtlCreateUserThread**. The latter two are undocumented. However, the general idea is to pass the address of **LoadLibrary** to one of these APIs so that a remote process has to execute the DLL on behalf of the malware.

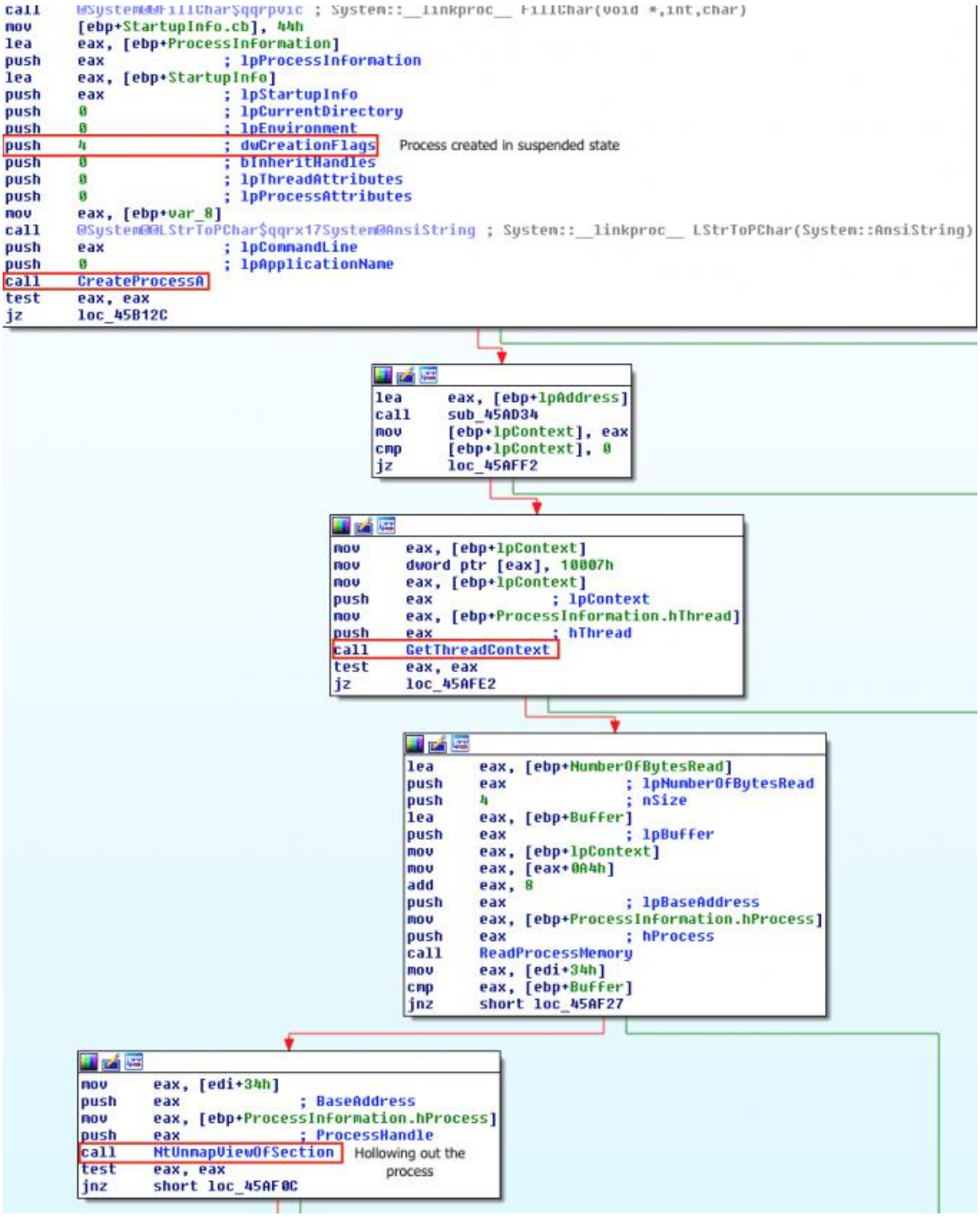
CreateRemoteThread is tracked and flagged by many security products. Further, it requires a malicious DLL on disk which could be detected. Considering that attackers are most commonly injecting code to evade defenses, sophisticated attackers probably will not use this approach.

2- PROCESS HOLLOWING (A.K.A PROCESS REPLACEMENT AND RUNPE):

Instead of injecting code into a host program (e.g., DLL injection), malware can perform a technique known as process hollowing. Process hollowing occurs when a malware un maps (hollows out) the legitimate code from memory of the target process, and overwrites the memory space of the target process (e.g., svchost.exe) with a malicious executable, and based on our analysis and reverse engineering this is the specific sequence APIs any malware will use to implement this technique as we can see below:

- **CreateProcessA:** Create a new process (in a suspended state).
- **NtUnmapViewOfSection** Remove the contents of a legitimate process from **memory.Putting** the process into a suspended state and removing its contents makes the process an empty shell.
- **VirtualAllocEx:** Assigns a new memory address into the hollow process
- **WriteProcessMemory:** Used to inject a new code in the hollow process
- **ResumeThread:** Used to resume the process flow.

The screenshot below displays the analysis and reverse engineering of a **Ransomware** named **Ransom.Cryak** performing this technique with the APIs as we explained.



The malware first creates a new process to host the malicious code in suspended mode. As shown in Figure, this is done by calling **CreateProcess** and setting the Process Creation Flag to CREATE_SUSPENDED (0x00000004). The primary thread of the new process is created in a suspended state, and does not run

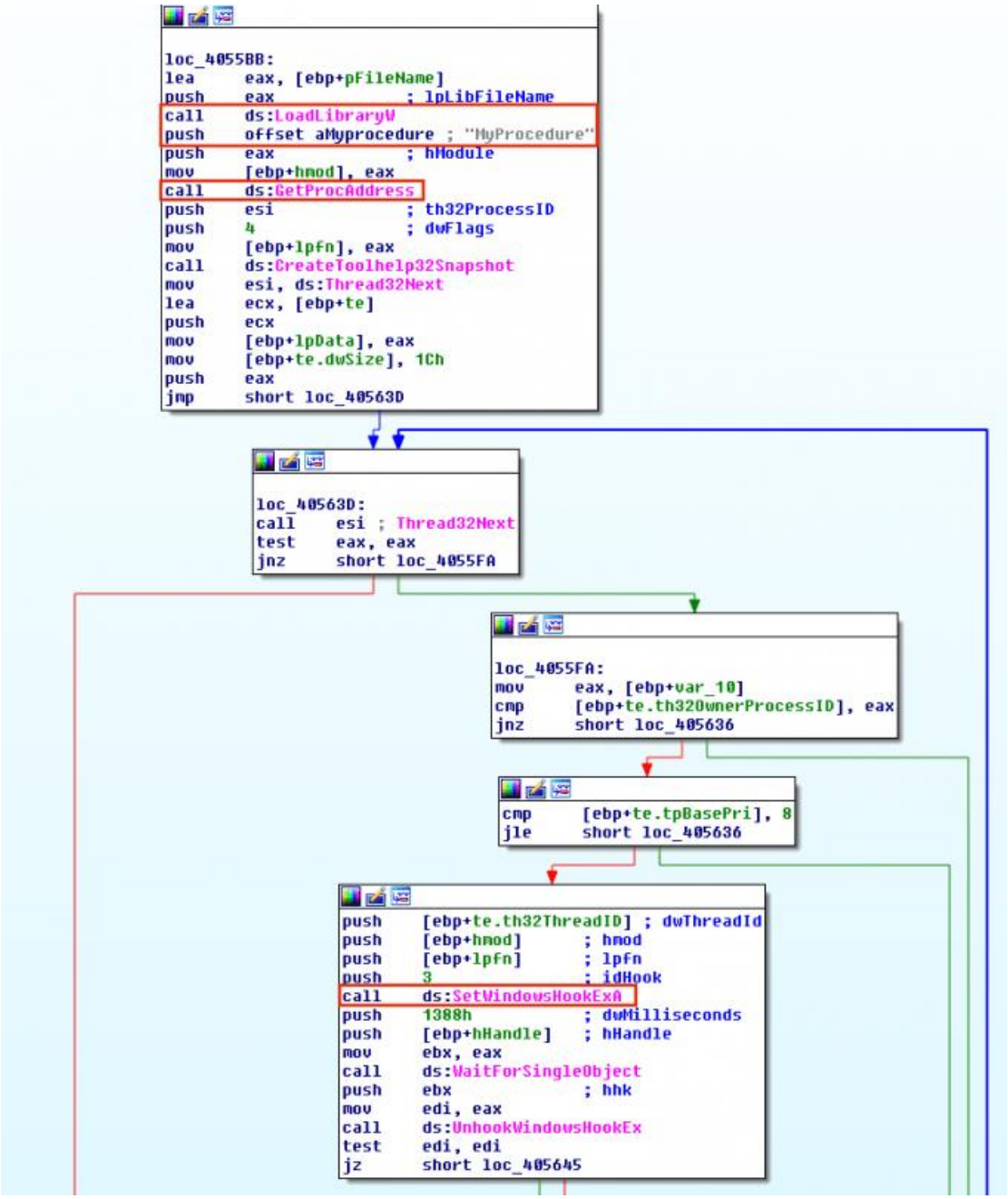
until the ResumeThread function is called. Next, the malware needs to swap out the contents of the legitimate file with its malicious payload. This is done by unmapping the memory of the target process by calling either **ZwUnmapViewOfSection** or NtUnmapViewOfSection. These two APIs basically release all memory pointed to by a section. Now that the memory is unmapped, the loader performs VirtualAllocEx to allocate new memory for the malware, and uses WriteProcessMemory to write each of the malware's sections to the target process space. The malware calls **SetThreadContext** to point the **entrypoint** to a new code section that it has written. At the end, the malware resumes the suspended thread by calling ResumeThread to take the process out of suspended state.

3- HOOK INJECTION VIA SETWINDOWSHOOKEX:

Hooking is a technique used to intercept function calls. Malware can leverage hooking functionality to have their malicious DLL loaded upon an event getting triggered in a specific thread. This is usually done by calling **SetWindowsHookEx** to install a hook routine into the hook chain, and based on our analysis and reverse engineering this is the specific sequence APIs any malware will use to implement this technique as we can see below:

- **LoadLibrary or LoadLibraryEx**
- **GetProcAddress**
- **SetWindowsHookEx**

The screenshot below displays the analysis and reverse engineering of a **Ransomware** named **Locky** Ransomware performing this technique with the APIs as we explained.



The **SetWindowsHookEx** function takes four arguments. The first argument is the type of event. The events reflect the range of hook types, and vary from pressing keys on the keyboard

(WH_KEYBOARD) to inputs to the mouse (WH_MOUSE), CBT, etc. The second argument is a pointer to the function the malware wants to invoke upon the event execution. The third argument is a module that contains the function. Thus, it is very common to see calls to **LoadLibrary** and **GetProcAddress** before calling **SetWindowsHookEx**. The last argument to this function is the thread with which the hook procedure is to be associated. If this value is set to zero all threads perform the action when the event is triggered. However, malware usually targets one thread for less noise, thus it is also possible to see calls **CreateToolhelp32Snapshot** and **Thread32Next** before **SetWindowsHookEx** to find and target a single thread. Once the DLL is injected, the malware executes its malicious code on behalf of the process that its thread Id was passed to **SetWindowsHookEx** function. In Figure, Locky Ransomware implements this technique.

4- THREAD EXECUTION HIJACKING (A.K.A SUSPEND, INJECT, AND RESUME (SIR))

This technique has some similarities to the process hollowing technique previously discussed. In thread execution hijacking, malware targets an existing thread of a process and avoids any noisy process or thread creations operations. Therefore, during analysis you will probably see calls to **CreateToolhelp32Snapshot** and **Thread32First** followed by **OpenThread**.

The screenshot below displays the analysis and reverse engineering of a trojan performing this technique with the APIs as we explained.

After getting a handle to the target thread, the malware puts the thread into suspended mode by calling **SuspendThread** to

perform its injection. The malware calls **VirtualAllocEx** and **WriteProcessMemory** to allocate memory and perform the code injection. The code can contain shellcode, the path to the malicious DLL, and the address of LoadLibrary.

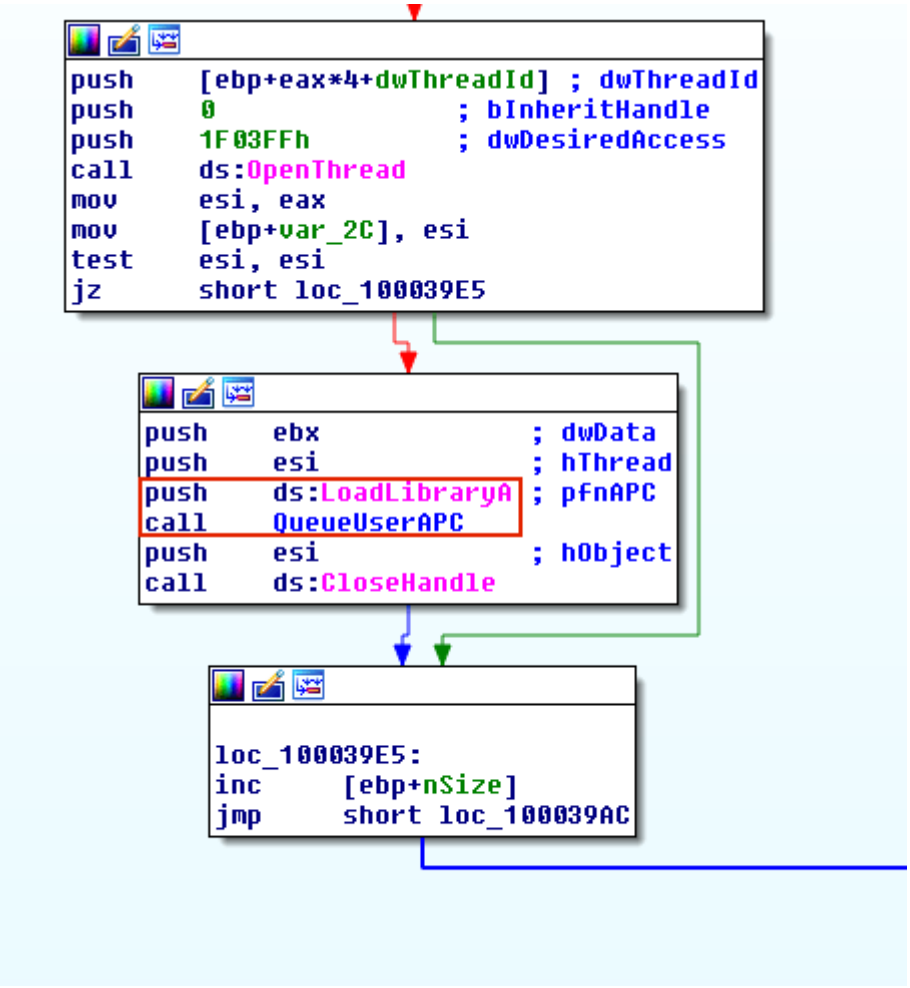
Figure illustrates a generic trojan using this technique. In order to hijack the execution of the thread, the malware modifies the EIP register (a register that contains the address of the next instruction) of the targeted thread by calling **SetThreadContext**. Afterwards, malware resumes the thread to execute the shellcode that it has written to the host process. From the attacker's perspective, the SIR approach can be problematic because suspending and resuming a thread in the middle of a system call can cause the system to crash. To avoid this, a more sophisticated malware would resume and retry later if the EIP register is within the range of **NTDLL.dll**.

5- APC INJECTION AND ATOMBOMBING:

Malware can take advantage of Asynchronous Procedure Calls (APC) to force another thread to execute their custom code by attaching it to the APC Queue of the target thread. Each thread has a queue of APCs which are waiting for execution upon the target thread entering alterable state, and based on our analysis and reverse engineering this is the specific sequence APIs any malware will use to implement this technique as we can see below:

- **OpenProcess** and **CreateTool32Snapshot**
- **Process32First** and **Thread32First**
- **Process32Next** and **Thread32Next**
- **VirtualAllocEx**, **VirtualFreeEx**, and **CloseHandle**
- **WriteProcessMemory**
- **QueueUserAPC/NtQueueApcThread**

The screenshot below displays the analysis and reverse engineering of a **Almanahe** win32 malware performing this technique with the APIs as we explained.



A thread enters an alertable state if it calls **SleepEx**, **SignalObjectAndWait**, **MsgWaitForMultipleObjectsEx**, **WaitForMultipleObjectsEx**, or **WaitForSingleObjectEx**

functions. The malware usually looks for any thread that is in an alterable state, and then calls **OpenThread** and **QueueUserAPC** to queue an APC to a thread. **QueueUserAPC** takes three arguments: 1) a handle to the target thread; 2) a pointer to the function that the malware wants to run; 3) and the parameter that is passed to the function pointer. In Figure, Amanahe malware first calls **OpenThread** to acquire a handle of another thread, and then calls **QueueUserAPC** with LoadLibraryA as the function pointer to inject its malicious DLL into another thread.

AtomBombing is a technique that was first introduced by enSilo research, and then used in Dridex V4. As we discussed in detail in a previous post, the technique also relies on APC injection. However, it uses atom tables for writing into memory of another process.

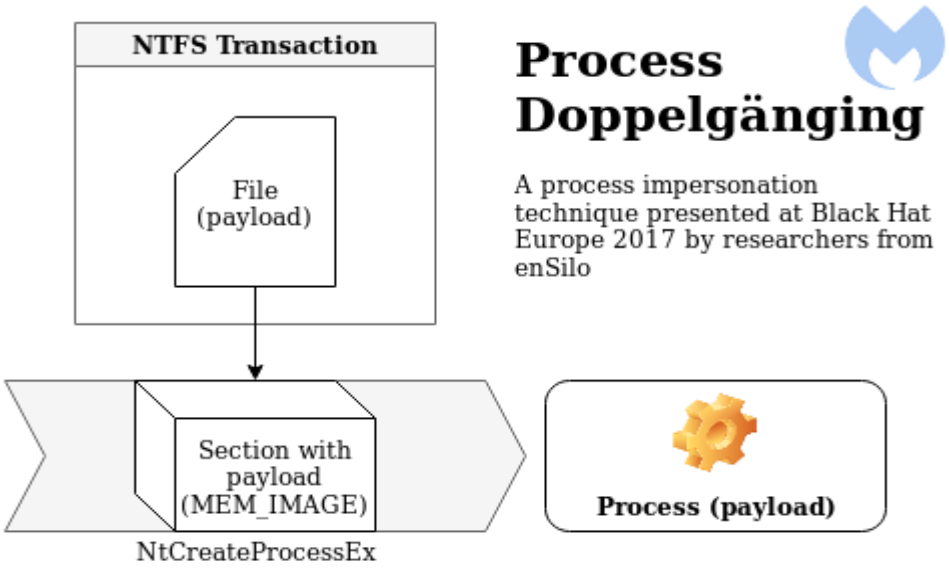
6- Process Doppelgänging Method APIs list:

Process Doppelgänging, one of the popular Code Injection techniques, was first announced by 2 security researchers working in enSilo company in BlackHat in 2017.

Process Doppelgänging is of great importance because it works successfully on all versions of Windows, including Windows 10. If it is similar to Process Hollowing, it has certain aspects that are strictly separated from Process Hollowing. Process Doppelgänging was frequently used by malware as it was difficult to detect by many AV products when it first appeared,

and based on our analysis and reverse engineering this is the specific sequence APIs any malware will use to implement this technique as we can see below:

- **CreateFileTransacted**, **WriteFIle**, and **NtCreateSection**
- **RollbackTransaction** and **NtCreateProcessEx**,
- **RtlCreateProcessParametersEx**, **VirtualAllocEx**,
- **WriteProcessMemory**, **NtCreateThreadEx**, and **NtResumeThread**



Process Hollowing first initiates the target process, then unmaps and injects the malicious code. Process Doppelgänging, on the other hand, writes the malicious code on the image before the process starts. This is actually the biggest difference between them.

Process Doppelgänging is implemented in 4 steps:

- **Transact** — Create a TxF transaction using a legitimate executable then overwrite the file with malicious code.

These changes will be isolated and only visible within the context of the transaction.

- **Load** — Create a shared section of memory and load the malicious executable.
- **Rollback** — Undo changes to original executable, effectively removing malicious code from the file system.
- **Animate** — Create a process from the tainted section of memory and initiate execution.

As a result, the process can start in an injected state even after the contents of the file are undone. For this reason, it will appear that there are no problems by many Anti-Virus products

7- INJECTION AND PERSISTENCE VIA REGISTRY MODIFICATION (E.G. APPINIT_DLLS, APPCERTDLLS, IFEO):

Appinit_DLL, AppCertDlls, and IFEO (Image File Execution Options) are all registry keys that malware uses for both injection and persistence. The entries are located at the following locations:

- HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls
- HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls
- HKLM\System\CurrentControlSet\Control\Session Manager\AppCertDlls
- HKLM\Software\Microsoft\Windows NT\currentversion\image file execution options

AppInit_DLLs:

Malware can insert the location of their malicious library under the **Appinit_Dlls** registry key to have another process load their library. Every library under this registry key is loaded into every process that loads **User32.dll**. **User32.dll** is a very common library used for storing graphical elements such as dialog boxes. Thus, when a malware modifies this subkey, the majority of processes will load the malicious library. Figure below demonstrates the trojan **Ginwui** relying on this approach for injection and persistence. It simply opens the **Appinit_Dlls** registry key by

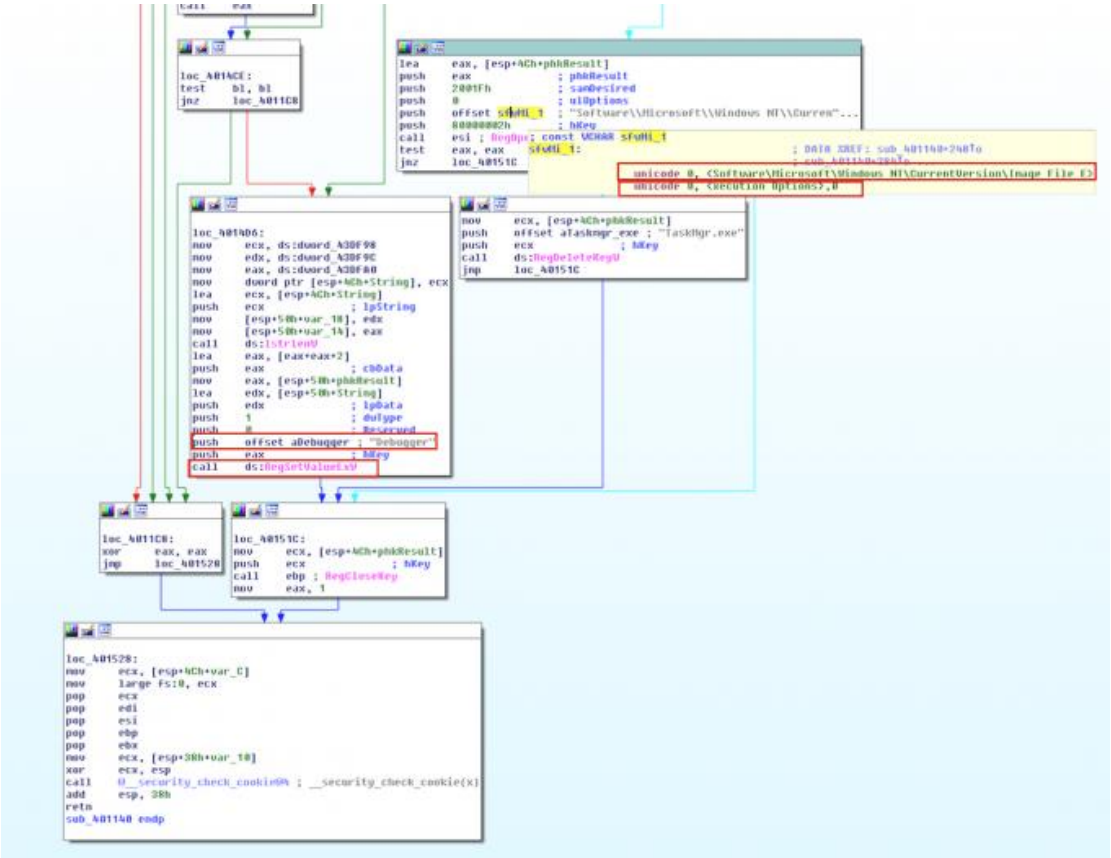
calling **RegCreateKeyEx**, and modifies its values by calling **RegSetValueEx**.

AppCertDlls:

This approach is very similar to the **AppInit_DLLs** approach, except that DLLs under this registry key are loaded into every process that calls the Win32 API functions **CreateProcess**, **CreateProcessAsUser**, **CreateProcessWithLogonW**, **CreateProcessWithTokenW**, and **WinExec**.

Image File Execution Options (IFEO):

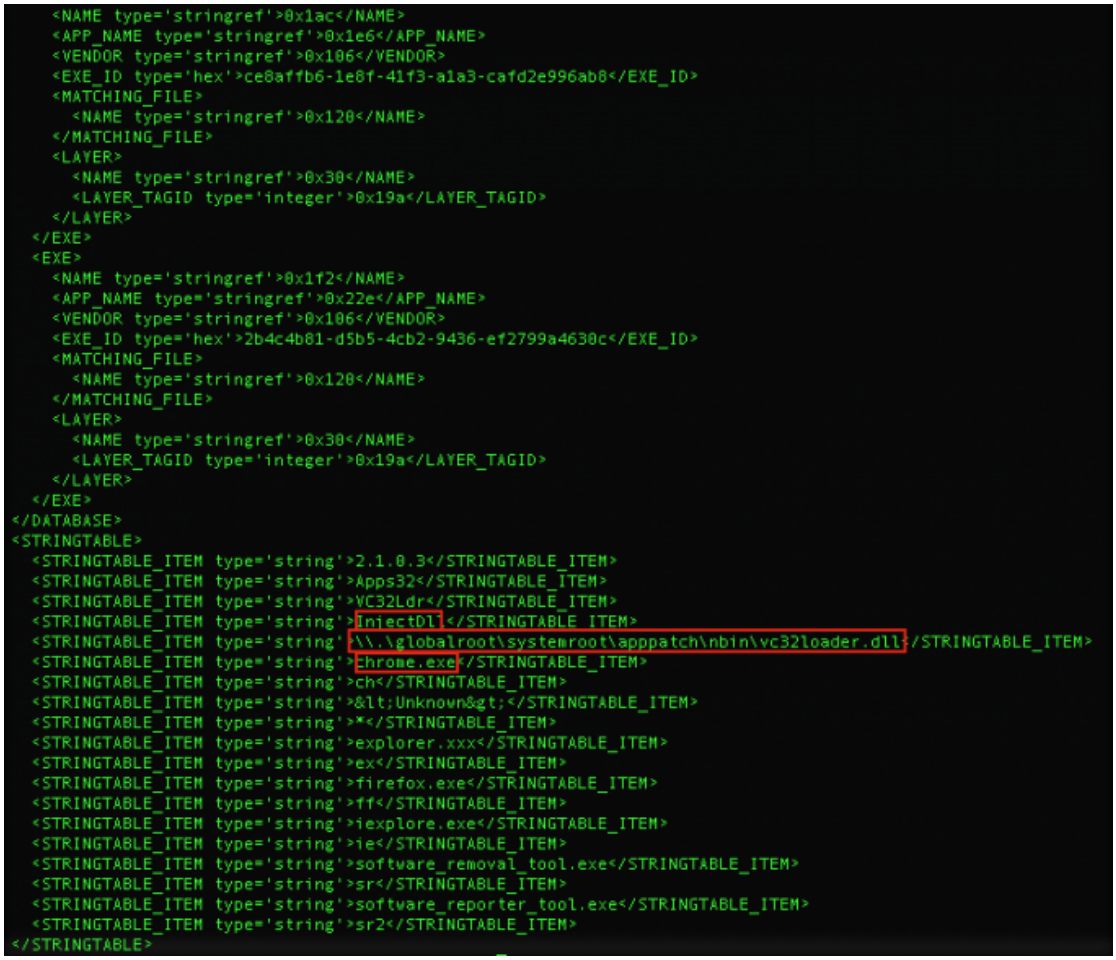
IFEO is typically used for debugging purposes. Developers can set the “Debugger Value” under this registry key to attach a program to another executable for debugging. Therefore, whenever the executable is launched the program that is attached to it will be launched. To use this feature, you can simply give the path to the debugger, and attach it to the executable that you want to analyze. Malware can modify this registry key to inject itself into the target executable. In Figure below, **Diztakun** trojan implements this technique by modifying the debugger value of Task Manager.



8- INJECTION USING SHIMS:

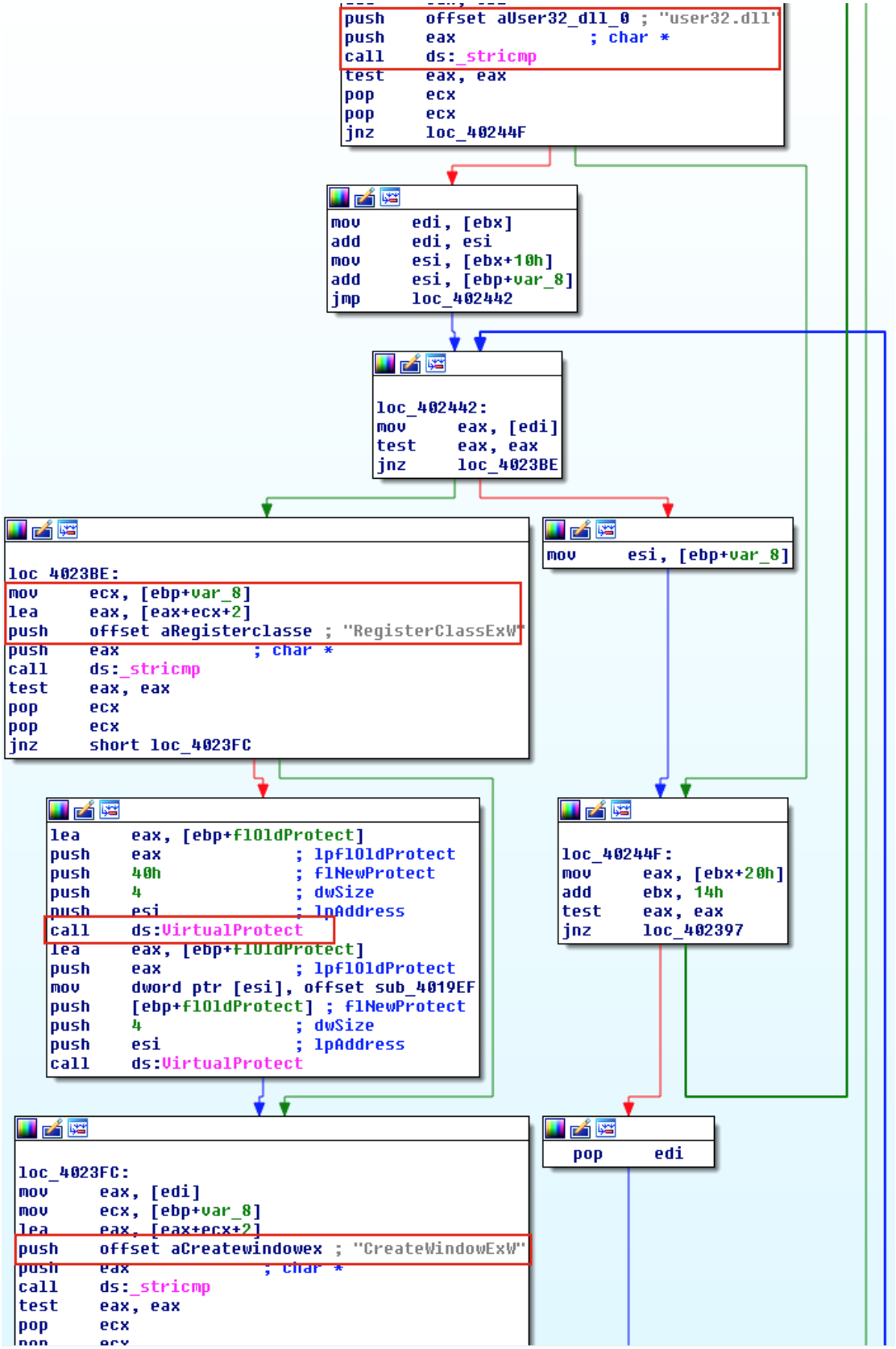
Microsoft provides Shims to developers mainly for backward compatibility. Shims allow developers to apply fixes to their programs without the need of rewriting code. By leveraging shims, developers can tell the operating system how to handle their application. Shims are essentially a way of hooking into APIs and targeting specific executables. Malware can take advantage of shims to target an executable for both persistence and injection. Windows runs the Shim Engine when it loads a binary to check for shimming databases in order to apply the appropriate fixes.

There are many fixes that can be applied, but malware's favorites are the ones that are somewhat security related (e.g., **DisableNX**, **DisableSEH**, **InjectDLL**, etc). To install a shimming database, malware can deploy various approaches. For example, one common approach is to simply execute sdbinst.exe, and point it to the malicious sdb file. In Figure below, an adware, "Search Protect by Conduit", uses a shim for persistence and injection. It performs an "**InjectDLL**" shim into Google Chrome to load vc32loader.dll. There are a few existing tools for analyzing sdb files, but for the analysis of the sdb listed below, I used python-sdb.



9- IAT HOOKING AND INLINE HOOKING (A.K.A USERLAND ROOTKITS):

IAT hooking and inline hooking are generally known as userland rootkits. IAT hooking is a technique that malware uses to change the import address table. When a legitimate application calls an API located in a DLL, the replaced function is executed instead of the original one. In contrast, with inline hooking, malware modifies the API function itself. In Figure below, the malware **FinFisher**, performs IAT hooking by modifying where the **CreateWindowEx** points.


push offset aUser32_dll_0 ; "user32.dll"
push eax ; char *
call ds:_stricmp
test eax, eax
pop ecx
pop ecx
jnz loc_40244F
mov edi, [ebx]
add edi, esi
mov esi, [ebx+10h]
add esi, [ebp+var_8]
jmp loc_402442
loc_402442:
mov eax, [edi]
test eax, eax
jnz loc_4023BE
loc_4023BE:
mov ecx, [ebp+var_8]
lea eax, [eax+ecx+2]
push offset aRegisterclasse ; "RegisterClassExW"
push eax ; char *
call ds:_stricmp
test eax, eax
pop ecx
pop ecx
jnz short loc_4023FC
mov esi, [ebp+var_8]
lea eax, [ebp+flOldProtect]
push eax ; lpflOldProtect
push 40h ; flNewProtect
push 4 ; dwSize
push esi ; lpAddress
call ds:VirtualProtect
lea eax, [ebp+flOldProtect]
push eax ; lpflOldProtect
mov dword ptr [esi], offset sub_4019EF
push [ebp+flOldProtect] ; flNewProtect
push 4 ; dwSize
push esi ; lpAddress
call ds:VirtualProtect
loc_40244F:
mov eax, [ebx+20h]
add ebx, 14h
test eax, eax
jnz loc_402397
loc_4023FC:
mov eax, [edi]
mov ecx, [ebp+var_8]
lea eax, [eax+ecx+2]
push offset aCreatewindowex ; "CreateWindowExW"
push eax ; char *
call ds:_stricmp
test eax, eax
pop ecx
pop edi

Ransomware APTs Attacks:

Ransomware tests	Experimental test	File Hash (SHA 256)
	CTP-Locker Ransomware	128a0f0cd5d10f864d5a0741ba25996b2bf74f580ac7918dec6516215801e39a
	Cerber Ransomware	cf262a9236eaf5230c219845823f36fd8c8e8b77ba882c34ce38a5087539cf71
	CrypFile2 Ransomware	a1e4693db6419eb5588f25d2b9f90db6c0e96e30a51fed5f0236cbdd49894e75
	CryptoMix Ransomware	a9a232cbff2c4347c1fcdeb1a3f1a6e45fbd4e93a107c6dd57fb8994df9d3bce
	CryptoShield Ransomware	d56fb2bdad7a50ab1f6ef76c67669452ed4da2bf865beafcf4956ab30bfa20fc
	GlobeImposter Ransomware	72ddceebe717992c1486a2d5a5e9e20ad331a98a146d2976c943c983e088f66b
	Gryphon Ransomware	933af0c69e1e622e5677e52c24545761c2843b3f52ea38e63bbe4786bfd6276e
	JAFF Ransomware	824901dd0b1660f00c3406cb888118c8a10f66e3258b5020f7ea289434618b13

	Mole Ransomware	c2e1770241fcc4b5c889fec68df024a6838e63e603f093715e3b468f9f31f67a
	NemucodAES Ransomware	482711b2f17870ddae316619ba2f487641e35ac4c099ae7e0ff4becd79e89faf (payload)
	Revenge Ransomware	8ab65ceef6b8a5d2d0c0fb3ddbe1c1756b5c224bafc8065c161424d63937721c
	TeslaCrypt Ransomware	200bc25fa093ce65f41baa1c3efe02dcc238b04cb57a6fc5ee87da1e04d6e168
	WannaCry Ransomware	ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa
	CrypMIC Ransomware	b2bcfc4c5d1d60f7ea4298d32dcfff303f4db4b1ba89a8b6d24b7ccfe883e45a

- **Analysis**

Of the 1262 calls to external functions across all experiments, 244 were present in ransomware which were further reduced to 209 calls by combining similar calls of ANSI and Unicode variants as shown in Table 7.

Calls	Grouped into
CopyFileA CopyFileExW CopyFileW	CopyFile [A|ExW|W]
CreateDirectoryA CreateDirectoryW	CreateDirectory [A|W]
…A …W …Ex …ExA …ExW	…[A|W|Ex|ExA|ExW]

The rationale for merging similar API calls is that Windows API calls such as **FindNextFileW** and **FindNextFileA** are essentially the same API call (the 'W' variant accepting Unicode and 'A' variant accepting ANSI coded input strings). Similarly, functions with Ex suffixes are generally newer with a different call pattern, however their base functionality is often quite similar.

The API calls were arranged into two-way contingency tables that plotted the observed frequency of each API call for each experimental test. We identified API calls of interest. Calls of interest were selected where the "API call's presence indicated ransomware regardless of call frequency" and "API calls with significantly higher- than-average call frequencies" statistics. We used Fisher exact tests to compare the

prevalence of each specific API call in the ransomware group to the normal baseline operations group. Calls with usage patterns that differed significantly ($p < 0.05$) between the two groups were identified.

- **Results**

An initial examination of the contingency table that compares all ransomware system calls to system calls made by non-malicious normal baseline operations show that ransomware used a small subset of all system calls logged during normal baseline operations. Comparing the frequency of all ransomware system calls to the frequency of system-calls in normal baseline operations shows that identification of ransomware can be done through call frequencies alone (chi-square; $p << 0.01$; 95% confidence level for significance testing). This is a reasonable expectation given the large data set and high variability in call frequencies and prevalence. The API calls which contributed most to the chi-square statistic were examined to determine what subset of calls could be used to indicate the presence of ransomware activity.

When we examine individual API calls more closely, we found that 18 Windows API calls where usage patterns (prevalence or call frequency) varied between ransomware and baseline normal operation differed significantly (Tables 8, 9 and 10). These API calls occur in significantly more ransomware strains (compared to baseline experiments), or at greater call frequencies ($p < 0.05$).

The interesting calls identified included:

- 8 API calls that existed only in ransomware at a significant level.
- 4 API calls that existed in both ransomware and normal operations, where the difference in utilization of the API call was statistically significant and more common in ransomware samples than in normal baseline operations.
- 6 API calls that existed in both ransomware and baseline normal operation and where the ransomware frequency count exceeded the baseline mean by more than three standard deviations (3σ).

As we can see in the table below Calls to Windows APIs (without considering call frequency) - ransomware vs normal baseline operations

	Windows API Call	**Count of ransomware samples used**	**Count of baseline samples used**	**Usage differs between ransomware and baseline (Fisher exact P-value)**
Present only in ransomware	InternetOpen	6	0	0.006
	CryptDeriveKey	5	0	0.017
	CryptDecodeObject	4	0	0.042
	CryptGenKey	4	0	0.042

	CryptImportPublicKeyInfo	4	0	0.042
	GetUserName	4	0	0.042
	NdrClientCall2	4	0	0.042
	socket	4	0	0.042
Used in more ransomware strains	_tailMerge_CRYPTSP_dll*	9	1	0.002
	CoCreateInstance	8	1	0.005
	SHWindowsPolicy	8	1	0.005
	GetFileType	10	4	0.027

And as we can see in the table below Calls to Windows APIs where ransomware call frequency exceeds baseline mean call frequency by more than 3 standard deviations.

	Windows API Call	**Count of ransomware samples using high ($\bar{x}$ + 3σ) frequency**	**Count of baseline samples using high ($\bar{x}$+ 3σ) frequency call rates**	**Significance (Fisher exact)**

		calls rates		
Used in ransomware at higher call frequency	CryptAcquireContext	7	0	0.002
	CloseHandle	6	0	0.006
	FindNextFile	6	0	0.006
	SetFilePointer	6	1	0.035
	GetFileSize	4	0	0.042
	SetFileAttributes	4	0	0.042

Finally, as we can see in the table below Calls to Windows specific APIs sequence categorized by ransomware strain

	Windows System Call	CTB-Locker	Cerber	CrypMIC	CryptFile2	CryptoMix	CryptoShield	GlobeImposter	Gryphon	JAFF	Mole	Revenge	*TeslaCrypt*	WannaCry	NemucodAES
Detected by call presence (exclusive to	InternetOpen				*	*	*			*	*	*			
	CryptDeriveKey				*	*	*		*			*			
	CryptDecodeObject		*						*		*	*			
	CryptGenKey						*			*	*	*			
	CryptImportPublicKeyInfo		*						*		*	*			

	GetUserName				*		*				*	*			
	NdrClientCall2					*	*		*			*			
	Socket		*	*		*	*								
Detected by call presence (not ransomware exclusive)	_tailMerge_CRYPTSP_dll (1 false positive)		*	*		*	*	*	*	*		*		*	
	CoCreateInstance (1 false positive)	*	*			*	*		*	*		*			*
	SHWindowsPolicy (4 false positivies)		*		*	*	*		*	*		*			*
	GetFileType (1 false positive)	*	*		*	*	*	*	*	*		*			*
Detected in ransomware through statistically high ($x^-+3\sigma$) call frequencies	CryptAcquireContext				*	*	*	*	*		*	*			
	CloseHandle			*		*	*	*	*			*			
	FindNextFile			*			*	*	*	*		*			
	SetFilePointer (1 false positive)		*		*	*	*	*				*			
	GetFileSize			*				*	*	*					
	SetFileAttributes		*				*				*	*			
	Count of calls capable of identifying ransomware	2	9	5	7	11	15	7	12	8	7	16	*0*	1	3
	* - Ransomware														

	Detected with System Call

The fisher-exact test of independence showed a very high level of certainty that the baseline versus ransomware samples differed through a systematic process, namely, that the presence of ransomware in the system and not in our baseline tests was not merely coincidental. For example, GetFileType is used more often in ransomware than in baseline samples runs (10 ransomware samples vs 4 baseline operations). However, due to the small sample sizes for both baseline and ransomware, the difference in the API usage by ransomware strains and the baseline tests within the significant range ($p=0.066 > 0.05$). As such, no specific API can be used for detecting ransomware. Rather, the APIs identified and reported in Table 10 can aid in the detection of ransomware strains that would otherwise remain undetected in a Win/32 standard operating environment. It must also be noted that none of these APIs are dangerous for a standard Win/32 operating environment. However, based on our findings, we found that calls to some of these APIs are more frequent than others during a ransomware infection.

For Process injection, it can be done by directly injecting code into another process, or by forcing a DLL to be loaded into another process as we can see in the table below

	Shellcode Injection	Forcing A DLL To Be Loaded	Sha256
1. DLL Injection		X	07b8f25e7b536f5b6f686c12d04edc37e11347c8acd5c53f98a174723078c365
2. PE Injection	X		ce8d7590182db2e51372a4a04d6a0927a65b2640739f9ec01cfd6c143b1110da
3. Process Hollowing	X		eae72d803bf67df22526f50fc7ab84d838efb2865c27aef1a61592b1c520d144
4. Thread Execution Hijacking	X		787cbc8a6d1bc58ea169e51e1ad029a637f22560660cc129ab8a099a745bd50e
5. Hook Injection		X	5d6ddb8458ee5ab99f3e7d9a21490ff4e5bc9808e18b9e20b6dc2c5b27927ba1
6. Registry Modification		X	9f10ec2786a10971eddc919a5e87a927c652e1655ddbbae72d376856d30fa27c
7. APC Injection		X	f74399cc0be275376dad23151e3d0c2e2a1c966e6db6a695a05ec1a30551c0ad
8. Shell Tray Window Injection	X		5e56a3c4d4c304ee6278df0b32afb62bd0dd01e2a9894ad007f4cc5f873ab5cf
9. Shim Injection		X	6d5048baf2c3bba85adc9ac5ffd96b21c9a27d76003c4aa657157978d7437a20
10. IAT and Inline Hooking	X	X	f827c92fbe832db3f09f47fe0dcaafd89b40c7064ab90833a1f418f2d1e75e8e

HTTP C&C Traffic APIs:

If C&C is implemented in malware, it will keep a constant contact (in HTTP) with the C2 server, due to its stateless nature. To initiate an HTTP session, the following APIs can be used:

- **InternetOpen()**

Example:

- **InternetOpen**(USER_AGENT,INTERNET_OPEN_TYPE_PROXY,argv[2],0,0);]
- **InternetConnect()** – URL Input to build HTTP request, the following APIs are used:

  HttpOpenRequest(), **HttpAddRequestHeaders()**

For sending HTTP requests: **HTTPSendRequest()**

- For reading response, which means that the malware may be reading the response: **InternetReadFile()**.

KEYSTROKES Loggers:

- Keyboard keystrokes can be logged either by polling Keyboard state or by using API hook of keyboard related events, such as:
- **GetAsyncKeyState**() -> This function polls the state of keys on the keyboard.
- **GetKeyState**() -> API call (eg: check if the shift key is pressed)
- **SetWindowsHookExA**() -> Used to retrieve WH_KEYBOARD and WH_MOUSE movement.

Network Traffic Monitor APIs:

- **WSASocket() or socket()** -> Used to initiate a raw socket
- **bind()** -> bind a particular socket to a NIC
- **WSAIoctl() or ioctlsocket()** -> Change the mode of a NIC into Promiscuous mode.(SIO_RECVALL – parameter tells OS to put the n/w card in promiscuous mode).

Dropper APIs:

Droppers may embed files in its resources section, such as Payloads in the resources section '.rsrc'.

To manage resources, these APIs are used:

- **FindResource**
- **LoadResource**
- **SizeOfResource**
- and **LockResource**

Extract and Build the Data-Set:

We build a python script to extract the specific features (APIs) as we explained from 2000 APTs Attacks, Malware samples and benign samples, as we can see in the figures below this is the script we used to extract the specific features (APIs), compare and filter it with our APIs list we created based on our analysis, then we converted it to ASCII code and saved it in the excel sheet to build our Data-Set.

```
import pefile
import os
import hashlib
import xlsxwriter
```

```
# iterate throw files
PATH = "C:/Users/infected/Downloads/sample"
for FILE in os.listdir(PATH):
    current_file = os.path.join(PATH, FILE)
    pe = pefile.PE(current_file)

    #calculate file hash and store it
    with open(current_file, "rb") as file_to_check:
        # read contents of the file
        data = file_to_check.read()
        # pipe contents of the file through
        md5_returned = hashlib.md5(data).hexdigest()
        worksheet.write(row, 0, md5_returned)
        column = 1
        worksheet.write(row, column, 0)
```

```
#extract APIs
for entry in pe.DIRECTORY_ENTRY_IMPORT:
    for API in entry.imports:
        s = str(API.name)

#convert APIs name into ascii and store it
        x = ''.join(str(ord(c)) for c in s)
        # Store API ascii if it's in our list
        if search(API_list, x):
            column += 1
            worksheet.write(row, column, x)
row += 1
```

```
#filter APIs
def search(list, API_name):
    for i in range(len(list)):
        if list[i] == API_name:
            return True
    return False
```

```
# Open XLSX file for writing
file_name = "c:/dataset/data.xlsx"
workbook = xlsxwriter.Workbook(file_name)
bold = workbook.add_format({'bold': True})
worksheet = workbook.add_worksheet()
```

we have been able to extract our specific features and build our Data-Set.

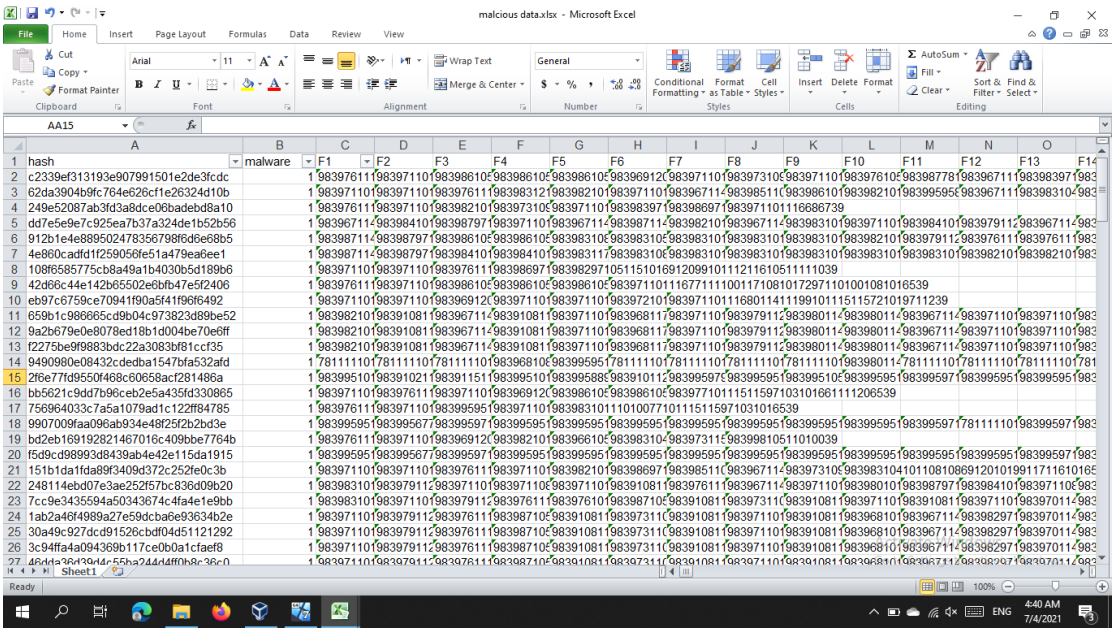
FEATURES:

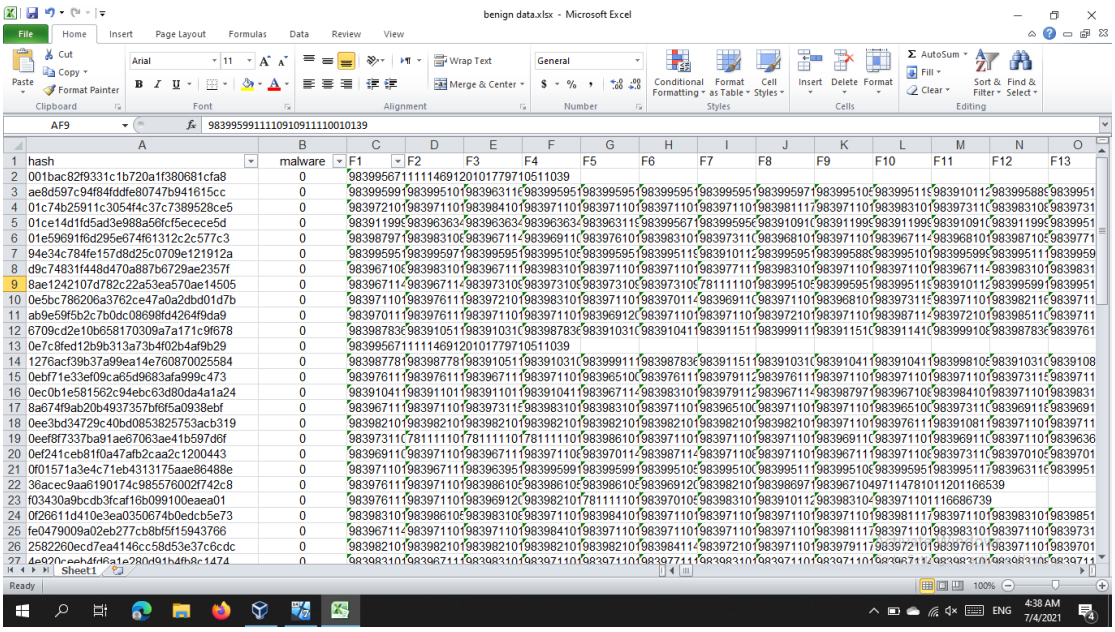
- **Column** name: hash
- Description: MD5 hash of the example
- Type: 32 bytes string

- **Column** name: F1 … F100
- Description: API call
- Type: ASCII code

- **Column** name: malware
- Description: Class
- Type: Integer: 0 (Goodware) or 1 (Malware)

we can see our Data-Set in the two figures below





now let's build a machine learning model that can detect a malicious binary activities based on our specific sequence of APIs (Data-set) techniques into the memory.

5.1.2 Artificial intelligence phase

Sample machine learning model

1) Import libraries and dependencies

We imported all libraries we need and dependencies to make project run in a good way.

```
# Import The Libraries

import numpy as np
import pandas as pd
import seaborn as sns
import pickle as pck
import matplotlib.pyplot as plt

import sklearn.ensemble as ske
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
%matplotlib inline

from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier


from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
```

2) Reading dataset

```
df = pd.read_csv('dynamic_api_call_sequence_per_malware_100_0_306.csv', sep= ',')
test_df = pd.read_csv('Malware_Samples.csv', sep = ',')
```

```
df.head()
```

	hash	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	...	t_91	t_92	t_93	t_94	t_95	t_96	t_97	t_98	t_99	malware
0	071e8c3f8922e186e57548cd4c703a5d	112	274	158	215	274	158	215	298	76	...	71	297	135	171	215	35	208	56	71	1
1	33f8e6d08a6aae939f25a8e0d63dd523	82	208	187	208	172	117	172	117	172	...	81	240	117	71	297	135	171	215	35	1
2	b68abd064e975e1c6d5f25e748663076	16	110	240	117	240	117	240	117	240	...	65	112	123	65	112	123	65	113	112	1
3	72049be7bd30ea61297ea624ae198067	82	208	187	208	172	117	172	117	172	...	208	302	208	302	187	208	302	228	302	1
4	c9b3700a77facf29172f32df6bc77f48	82	240	117	240	117	240	117	240	117	...	209	260	40	209	260	141	260	141	260	1

5 rows × 102 columns

```
test_df
```

	hash	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	...	t_90	t_91	t_92	t_93	t_94	t_95	t_96	t_97	t_98	t_99
0	93aa281c119d7d0058e307842c614dfd	240	117	240	117	240	117	240	117	240	...	208	240	117	240	117	240	117	240	117	240
1	ebabdff7c9c321662844d30d47b96af9	82	208	187	208	172	117	172	208	16	...	117	274	215	106	171	260	141	65	240	117
2	ca128ac7808532a517045aec7ae09265	112	274	158	215	274	158	215	298	76	...	117	71	297	135	171	215	35	208	56	71
3	06b43cb00b61be55b6d100b15edfbc39	172	117	172	117	172	117	198	208	260	...	60	81	260	172	117	25	240	117	71	297
4	7b38143a929cfd306a29b33da166e521	82	16	35	240	117	86	208	86	31	...	60	81	25	60	81	25	60	81	25	60
5	c124a717590ec531d658a4abcc367fee	286	172	117	275	208	187	208	240	275	...	260	141	65	260	215	240	117	71	297	135

6 rows × 101 columns

3) Reduce dimensionality

The code below to reduce the dimensionality of the features We use Extract Trees Classifier and Select from Model The most important features that have a strong relationship with the target column (Malware or Goodware).

```
# Feature selection using Trees Classifier
fsel = ske.ExtraTreesClassifier().fit(X, y)
model = SelectFromModel(fsel, prefit=True)
X = model.transform(X)
nb_features = X.shape[1]
```

```
features = []

print('%i features identified as important:' % nb_features)
```

```
31 features identified as important:
```

```
indices = np.argsort(fsel.feature_importances_)[::-1][:nb_features]

for f in range(nb_features):
    print("%d. feature %s (%f)" % (f + 1, df.columns[2+indices[f]], fsel.feature_importances_[indices[f]]))
```

```
1. feature t_2 (0.044445)
2. feature t_77 (0.031929)
3. feature t_91 (0.020506)
4. feature t_1 (0.018732)
5. feature t_99 (0.017880)
6. feature t_80 (0.017809)
7. feature t_79 (0.017577)
8. feature t_90 (0.016558)
9. feature t_76 (0.016181)
10. feature t_3 (0.014531)
11. feature t_24 (0.013821)
12. feature t_94 (0.013649)
13. feature t_64 (0.013627)
14. feature t_21 (0.012499)
15. feature t_60 (0.012317)
16. feature t_47 (0.011796)
17. feature t_26 (0.011651)
18. feature t_69 (0.011573)
19. feature t_62 (0.011397)
20. feature t_58 (0.011220)
21. feature t_95 (0.011158)
22. feature t_53 (0.010894)
23. feature t_29 (0.010503)
24. feature t_81 (0.010468)
25. feature t_13 (0.010373)
26. feature t_84 (0.010324)
27. feature t_18 (0.010313)
28. feature t_88 (0.010261)
29. feature t_15 (0.010235)
30. feature t_82 (0.010115)
31. feature t_86 (0.010041)
```

```
# Take care of the feature order

for f in sorted(np.argsort(fsel.feature_importances_)[::-1][:nb_features]):
    features.append(df.columns[2+f])
```

4) Splitting the data to train and test set

Then we split the dataset to train set and validation set:

Train set to train the model on it.

Validation set to evaluate the performance of the model.

```
#SPLITING DATA

X = df.drop(['hash', 'malware'], axis = 1).values
y = df['malware'].values
```

```
# Splitting the dataset into the Training set and Test set
#The target is Malware Column (0=Benign, 1=Malware)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 101)
```

```
print('X train shape: ', X_train.shape)
print('Y train shape: ', y_train.shape)
print('X test shape: ', X_test.shape)
print('Y test shape: ', y_test.shape)
```

```
X train shape:  (35100, 31)
Y train shape:  (35100,)
X test shape:  (8776, 31)
Y test shape:  (8776,)
```

5) building the model

We used 6 Algorithms to compare them and get the best one Algorithms are (Random Forest Classifier (RF), Extreme Gradient Boosting Classifier (XGBC), Gradient Boosting Classifier (GBC), K-Nearest Neighbors Classifier (KNN), Support vector classifier (SVC), Multi-Layer Perceptron (MLP)).

Random Forest Classifier:

Random Forest Classifier

```
rf_clf= RandomForestClassifier(n_estimators = 100, random_state = 0, oob_score = True, max_depth = 16, max_features = 'sqrt')
rf_clf.fit(X_train, y_train)

y_pred = rf_clf.predict(X_test)
```

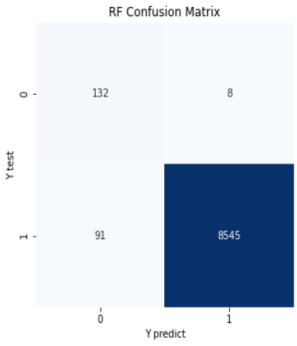
Confusion Matrix to evaluate the model:

Confusion Matrix is commonly used for a summarization of prediction results on a classification problem. The number of correct and incorrect predictions is summarized with counting values and each value broken down for each class. Each of them is the key to the confusion matrix. It shows the classification model is confused when it makes predictions, at this point in here it gives us insight not only into the errors being made by a classifier but also show the types of errors that are being made.

Accuracy score (training): 0.994

Accuracy score (validation): 0.989

```
rf_cm = confusion_matrix(y_test, y_pred)
f, ax = plt.subplots(figsize=(5,5))
sns.heatmap(confusion_matrix(y_pred, y_test), annot = True, fmt = 'd', cmap = plt.cm.Blues, cbar = False)
plt.title('RF Confusion Matrix')
plt.xlabel('Y predict')
plt.ylabel('Y test')
plt.show()

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of Random Forest Classifier: %.2f%%" % (accuracy * 100.0))
```




```
Accuracy of Random Forest Classifier: 98.87%
```

```
print("Accuracy score (training): {0:.3f}".format(rf_clf.score(X_train, y_train)))
print("Accuracy score (validation): {0:.3f}".format(rf_clf.score(X_test, y_test)))
```

```
Accuracy score (training): 0.994
Accuracy score (validation): 0.989
```

XGB Classifier:

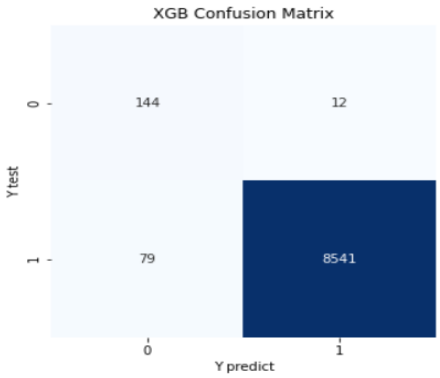
```
# XGBClassifier

xgb_clf = XGBClassifier()
xgb_clf.fit(X_train, y_train)

y_pred2 = xgb_clf.predict(X_test)
```

Confusion Matrix to evaluate the model:

Accuracy score (training): 0.999

Accuracy score (validation): 0.9905.2.5.3

```
]: xgb_cm = confusion_matrix(y_test, y_pred2)
f, ax = plt.subplots(figsize=(5,5))
sns.heatmap(confusion_matrix(y_pred2, y_test), annot = True, fmt = 'd', cmap = plt.cm.Blues, cbar = False)
plt.title('XGB Confusion Matrix')
plt.xlabel('Y predict')
plt.ylabel('Y test')
plt.show()

accuracy = accuracy_score(y_test, y_pred2)
print("Accuracy of XGBClassifier: %.2f%%" % (accuracy * 100.0))
```




```
Accuracy of XGBClassifier: 98.96%
```

```
]: print("Accuracy score (training): {0:.3f}".format(xgb_clf.score(X_train, y_train)))
print("Accuracy score (validation): {0:.3f}".format(xgb_clf.score(X_test, y_test)))
```

```
Accuracy score (training): 0.999
Accuracy score (validation): 0.990
```

Gradient Boosting Classifier:

Gradient Boosting Classifier

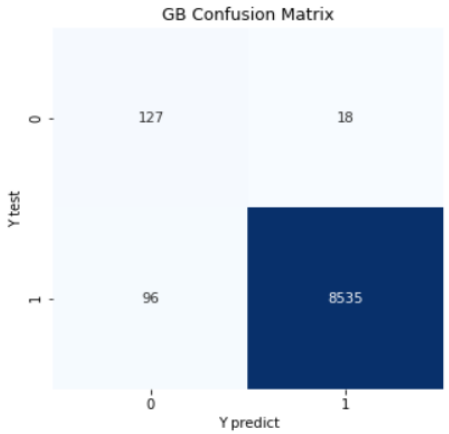
```
gb_clf = GradientBoostingClassifier()
gb_clf.fit(X_train, y_train)

y_pred3 = gb_clf.predict(X_test)
```

Confusion Matrix to evaluate the model:

Accuracy score (training): 0.991

Accuracy score (validation): 0.987

```
gb_cm = confusion_matrix(y_test, y_pred3)
f, ax = plt.subplots(figsize=(5,5))
sns.heatmap(confusion_matrix(y_pred3, y_test), annot = True, fmt = 'd', cmap = plt.cm.Blues, cbar = False)
plt.title('GB Confusion Matrix')
plt.xlabel('Y predict')
plt.ylabel('Y test')
plt.show()

accuracy = accuracy_score(y_test, y_pred3)
print("Accuracy of Gradient Boosting Classifier: %.2f%%" % (accuracy * 100.0))
```




```
Accuracy of Gradient Boosting Classifier: 98.70%
```

```
print("Accuracy score (training): {0:.3f}".format(gb_clf.score(X_train, y_train)))
print("Accuracy score (validation): {0:.3f}".format(gb_clf.score(X_test, y_test)))
```

```
Accuracy score (training): 0.991
Accuracy score (validation): 0.987
```

K-Nearest Neighbors Classifier:

KNeighborsClassifier

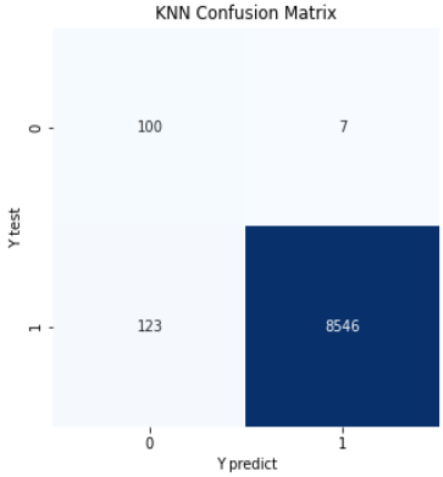
```
knn_clf = KNeighborsClassifier(n_neighbors=5)
knn_clf.fit(X_train, y_train)

y_pred4 = knn_clf.predict(X_test)
```

Confusion Matrix to evaluate the model:

Accuracy score (training): 0.987

Accuracy score (validation): 0.985

```
knn_cm = confusion_matrix(y_test, y_pred4)
f, ax = plt.subplots(figsize=(5,5))
sns.heatmap(confusion_matrix(y_pred4, y_test), annot = True, fmt = 'd', cmap = plt.cm.Blues, cbar = False)
plt.title('KNN Confusion Matrix')
plt.xlabel('Y predict')
plt.ylabel('Y test')
plt.show()

accuracy = accuracy_score(y_test, y_pred4)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```




```
Accuracy: 98.52%
```

```
print("Accuracy score (training): {0:.3f}".format(knn_clf.score(X_train, y_train)))
print("Accuracy score (validation): {0:.3f}".format(knn_clf.score(X_test, y_test)))
```

```
Accuracy score (training): 0.987
Accuracy score (validation): 0.985
```

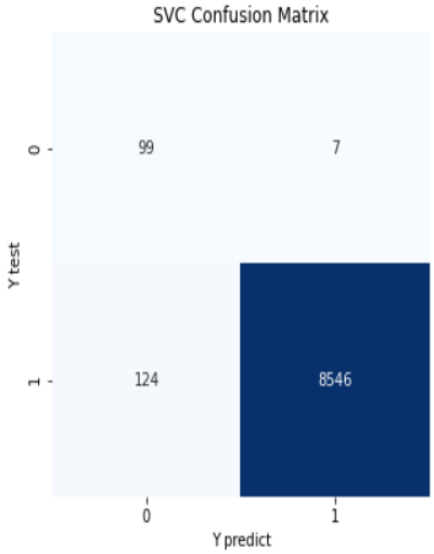
Support Vector Classifier:

SVC

```
sv_clf = SVC()
sv_clf.fit(X_train, y_train)

y_pred5 = sv_clf.predict(X_test)
```

Confusion Matrix to evaluate the model:

```
sv_cm = confusion_matrix(y_test, y_pred5)
f, ax = plt.subplots(figsize=(5,5))
sns.heatmap(confusion_matrix(y_pred5, y_test), annot = True, fmt = 'd', cmap = plt.cm.Blues, cbar = False)
plt.title('SVC Confusion Matrix')
plt.xlabel('Y predict')
plt.ylabel('Y test')
plt.show()

accuracy = accuracy_score(y_test, y_pred5)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```




```
Accuracy: 98.51%
```

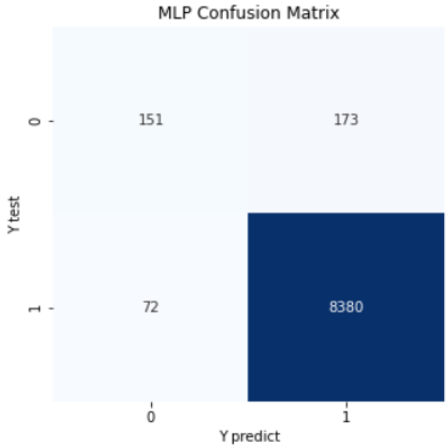
Multi-Layer Perceptron:

MLP Classifier

```
# We define the model
mlp_clf = MLPClassifier(hidden_layer_sizes=(100,100,100),max_iter=1000, random_state=42)

# We train model
mlp_clf.fit(X_train, y_train)


# We predict target values
y_pred6 = mlp_clf.predict(X_test)
```

Confusion Matrix to evaluate the model:

```
mlp_cm = confusion_matrix(y_test, y_pred6)
f, ax = plt.subplots(figsize=(5,5))
sns.heatmap(confusion_matrix(y_pred6, y_test), annot = True, fmt = 'd', cmap = plt.cm.Blues, cbar = False)
plt.title('MLP Confusion Matrix')
plt.xlabel('Y predict')
plt.ylabel('Y test')
plt.show()

accuracy = accuracy_score(y_test, y_pred6)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```




```
Accuracy: 97.21%
```

Average-precision-recall score:

```
from sklearn.metrics import average_precision_score
average_precision1 = average_precision_score(y_test, y_pred)
average_precision2 = average_precision_score(y_test, y_pred2)
average_precision3 = average_precision_score(y_test, y_pred3)
average_precision4 = average_precision_score(y_test, y_pred4)
average_precision5 = average_precision_score(y_test, y_pred5)
average_precision6 = average_precision_score(y_test, y_pred6)

print('Average precision-recall score for Random Forest Classifier : {0:0.2f}'.format(
      average_precision1))
print('Average precision-recall score for XGBClassifier : {0:0.2f}'.format(
      average_precision2))
print('Average precision-recall score for Gradiant Boosting Classifier : {0:0.2f}'.format(
      average_precision3))
print('Average precision-recall score for KNeighbors Classifier : {0:0.2f}'.format(
      average_precision4))
print('Average precision-recall score for SVC : {0:0.2f}'.format(
      average_precision5))
print('Average precision-recall score for MLP Classifier : {0:0.2f}'.format(
      average_precision6))
```

```
Average precision-recall score for Random Forest Classifier : 0.99
Average precision-recall score for XGBClassifier : 0.99
Average precision-recall score for Gradiant Boosting Classifier : 0.99
Average precision-recall score for KNeighbors Classifier : 0.99
Average precision-recall score for SVC : 0.99
Average precision-recall score for MLP Classifier : 0.99
```

Algorithms Comparison to get the winner:

Winner algorithm is XGB with a 98.963081 % success which we will use in the project.

```
#Algorithm comparison
models = {
    'RF' : rf_clf, 'XGB': xgb_clf, 'GB' : gb_clf, 'KNN' : knn_clf, 'SV' : sv_clf, 'MLP' : mlp_clf
}

results = {}
print("\nNow testing algorithms")

# Fitting Classification algorithms to the Training set
for model in models:
    clf = models[model]
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)
    print("%s : %f %%" % (model, score*100))
    results[model] = score

winner = max(results, key=results.get)
print('\nWinner algorithm is %s with a %f %% success' % (winner, results[winner]*100))
```

```
Now testing algorithms
RF : 98.871923 %
[20:50:19] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/learner.cc:1095: Starting in XGBoost 1.3.0, the default evaluation metric used with the objective 'binary:log
istic' was changed from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore the old behavior.
C:\ProgramData\Anaconda3\lib\site-packages\xgboost\sklearn.py:1146: UserWarning: The use of label encoder in XGBClassifier is deprecated and will be removed in a future release. To remove this w
arning, do the following: 1) Pass option use_label_encoder=False when constructing XGBClassifier object; and 2) Encode your labels (y) as integers starting with 0, i.e. 0, 1, 2, ..., [num_class
- 1].
  warnings.warn(label_encoder_deprecation_msg, UserWarning)
XGB : 98.963081 %
GB : 98.723792 %
KNN : 98.518687 %
SV : 98.507293 %
MLP : 97.208295 %

Winner algorithm is XGB with a 98.963081 % success
```

Save the algorithm and the feature list for later predictions:

```
# Save the algorithm and the feature list for later predictions
import pickle, joblib

print('Saving algorithm and feature list in classifier directory...')
joblib.dump(models[winner], 'classifier/classifier.pkl')
open('classifier/features.pkl', 'wb').write(pickle.dumps(features))
print('Saved')
```

```
Saving algorithm and feature list in classifier directory...
Saved
```

5.3 Summary

in this chapter we explained our detection methodology and our machine learning model based on our analysis of many APTs attacks and reverse engineering of many malware samples, our features based on extracting advanced specific sequence of APIs to detect malicious binary activities like Process Hollowing, Process Doppelgänging, Reflective PE Injection, Thread Execution Hijacking, DLL injection, Anti-Debugger, Anti-VM, API Hooking, malicious C&C traffic, HTTP C&C Traffic APIs, Network Traffic Monitor, Ransomware Encryption Algorithms APIs, KEYSTROKES Loggers APIs.

We extracted our features from 2000 samples , created a Data-set and collect another Data-Set from Kaggle, then we trained our machine learning model to detect malicious activities based on this features (specific APIs sequence).

Chapter 6

Conclusion and Future Work

Chapter 6

Conclusion and Future Work

6.1 Conclusion:

In this research, we have proposed a malware detection module based on advanced specific sequence of APIs techniques into Memory from APTs attacks, Malware samples and machine learning to detect malicious activities like Process Hollowing, Process Doppelgänging, Reflective PE Injection, Thread Execution Hijacking, Dll injection, Anti-Debugger, Anti-VM, API Hooking, HTTP C&C Traffic APIs, Network Traffic Monitor APIs, Ransomware Encryption Algorithms APIs, KEYSTROKES Loggers APIs. This module can be implemented at enterprise gateway level to act as a central antivirus engine to supplement antiviruses present on end user computers. This will not only easily detect known viruses, but act as a knowledge that will detect newer forms of harmful files. While a costly model requiring costly infrastructure, it can help in protecting invaluable enterprise data from security threat, and prevent immense financial damage.

6.2 Future Research:

In the future, our research will be the kernel of a malware detection engine for edge and cloud computing threats, we plan to add new

features “malicious encrypted TLS traffic metadata” and then introduce our malware detection engine into the world of the IOT, connect it to ESP32 and monitor edge computing connections to detect any threat or data theft.

References:

1. https://attack.mitre.org/techniques/T1186/

2. https://www.blackhat.com/docs/eu-17/materials/eu-17-Liberman-Lost-In-Transaction-Process-Doppelganging.pdf

3. https://medium.com/@alpinoacademy/understanding-and-detecting-dll-1nj3ct0n-process-hollowing-fcd87676d36b

4. http://halilozturkci.com/adli-bilisim-zararli-kod-analizinde-dll-injection-ve-process-hollowing-tespiti/

5. https://blog.malwarebytes.com/threat-analysis/2018/08/process-doppelganging-meets-process-hollowing_osiris/

6. https://kaganisildak.com/2019/02/10/process-doppelganging/

7. https://cysinfo.com/detecting-deceptive-hollowing-techniques/

8. https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

9. https://www.amazon.com/Professional-Assembly-Language-Richard-Blum/dp/0764579010

10. https://beginners.re/

11. https://www.amazon.com/Practical-Reverse-Engineering-Reversing-Obfuscation/dp/1118787315

12. https://www.amazon.com/Art-Memory-Forensics-Detecting-Malware/dp/1118825098/ref=pd_sbs_5/143-5110202-0524039?pd_rd_w=Krlde&pf_rd_p=43345e03-9e2a-47c0-9b70-a50aa5ecbd5c&pf_rd_r=QER60MBHP3WN9FES78FH&pd_rd_r=99ede023-6bce-4fc9-9c20-268d71c5d49c&pd_rd_wg=HGygp&pd_rd_i=1118825098&psc=1

13. https://www.amazon.com/Learning-Malware-Analysis-techniques-investigate-ebook/dp/B073D49Q6W

14. https://www.amazon.com/Malware-Data-Science-Detection-Attribution/dp/1593278594

15. https://core.ac.uk/download/pdf/159235636.pdf

16. https://elearnsecurity.com/product/ecmap-certification/

17. https://www.sans.org/cyber-security-courses/reverse-engineering-malware-malware-analysis-tools-techniques/

18. https://drive.google.com/drive/u/3/folders/1grlkHlPtwoLiYiNFzf1-LD0HjDwtuOUl

19. https://www.researchgate.net/figure/API-based-ransomware-detection-system-API-RDS-services_fig1_334982816

20. https://ntcore.com/

21. https://www.winitor.com/

22. https://www.secpod.com/blog/introduction-to-ida-pro/

23. https://www.nsa.gov/resources/everyone/ghidra/

24. https://www.varonis.com/blog/how-to-use-x64dbg/

25. https://www.datacamp.com/tracks/data-scientist-with-python

26. https://www.udemy.com/course/machine-learning-arabic/

27. https://www.coursera.org/professional-certificates/ibm-data-analyst

28. https://www.credly.com/badges/3ea5bb38-1a2d-4c66-805d-05cdb0e1da55?source=linked_in_profile

29. https://www.youracclaim.com/badges/a1010bd3-900b-47bd-93a8-cc3f7ae8a9da?source=linked_in_profile

30. https://www.credly.com/badges/d0c5ace7-8cc8-42c6-9135-500e0dbcdce3?source=linked_in_profile

31. https://www.kaggle.com/ang3loliveira/malware-analysis-datasets-api-call-sequences/code