



University of
Zurich^{UZH}

Design and Implementation of a System for Reproducible Machine and Deep Learning Models

Viachaslau Berasneu
Zürich, Switzerland
Student ID: 20-734-109

Supervisor: Dr. Alberto Huertas, Jan von der Assen
Date of Submission: September 13, 2023

Zusammenfassung

In den letzten Jahren sind kleine und mittlere Unternehmen (KMU) zunehmend von Technologie abhängig geworden, investieren jedoch weniger in die Cybersicherheit. Dies macht sie anfällig für Malware-Angriffe, die sich zunehmend auf Unternehmen anstelle von Einzelpersonen konzentrieren und erhebliche wirtschaftliche Auswirkungen haben. Dieses Projekt schlägt die Entwicklung und Implementierung eines Prototyp-Tools vor, das es ermöglicht, Maschinenlernmodelle innerhalb der SecBox-Sandbox-Umgebung zu trainieren, zu speichern und zu testen. Sowohl Klassifikations- als auch Anomalieerkennungsmodelle werden über Scikit-learn implementiert, um Vorhersagen über bekannte Malware-Typen (binäre und multiklassige Klassifikation) sowie die Erkennung von unbekannter Malware in Echtzeit während der Ausführung in der SecBox bereitzustellen. Die Modelle werden mit den Protokollen zur Ausführung von Systemaufrufen und Ressourcennutzung trainiert, die aus der SecBox verfügbar sind, und in geeignete Formate umgewandelt, indem datenbasierte und sequenzbasierte Datenvorverarbeitung verwendet wird. Die Reproduzierbarkeit der Modelle wird durch die Erstellung von Konfigurationsdateien mit Verweisen auf die Zufallsseed-Werte, die in der Schulung verwendeten Datensätze und andere Modellparameter sichergestellt, die zur erneuten Schulung desselben Modells verwendet werden können. Zur Bewertung und Vergleich der Modellleistung wird jeder Modelltyp in einem realistischen Szenario der Ausführung von Monti-Ransomware innerhalb der SecBox getestet, wobei eine Verwirrungsmatrix erstellt sowie Genauigkeits-, Präzisions-, Rückruf- und F1-Score-Metriken auf der Grundlage der Modellvorhersagen berechnet werden. Die Systemaufruf-Klassifikatormodelle zeigen die beste Leistung bei der Klassifizierung von Monti-Malware-Proben, und das Projekt schließt mit der Angabe mehrerer relevanter Forschungsbereiche ab, die weiter untersucht werden sollen.

Abstract

In recent years, small and midsize enterprises (SMEs) have become increasingly reliant on technology, but lag in terms of investment into cybersecurity. This renders them vulnerable to malware attacks, which are increasingly targeting companies rather than individuals, with great economic impact. This project proposes and implements a prototype tool, which allows for machine learning models to be trained, stored, and tested within the SecBox sandbox environment. Both classification and anomaly detection models are implemented through Scikit-learn, in order to provide predictions about known malware types (binary and multiclass classification), as well as detecting the presence of unseen malware in real-time during the SecBox execution. The models are trained using the system call and resource usage file execution logs available from the SecBox, which are transformed into suitable formats using frequency-based and sequence-based data preprocessing. Model reproducibility is ensured by generating configuration files with references to the random seeds, the datasets used in training, as well as other model parameters, which can be used to re-train the same model. To evaluate and compare model performance, each model type is tested in a realistic scenario of the execution of Monti ransomware within the SecBox, creating a confusion matrix as well as calculating the accuracy, precision, recall and F1-score metrics based on the model predictions. The system call classifier models are shown to have the best performance when classifying Monti malware samples, and the project is concluded by specifying several relevant research areas to be investigated further.

Acknowledgments


Dr. Alberto Huertas and Jan von der Assen, who supported and guided me on this project.

Prof. Dr. Burkhard Stiller, for providing me the opportunity to write this thesis at the Communication Systems Group.

Declaration of Independence for Written Work

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, 13.09.2023



Signature of student

Contents

Zusammenfassung	i
Abstract	iii
Acknowledgments	v
Declaration of Independence	vii
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	2
1.3 Thesis Outline	2
2 Background	5
2.1 Sandboxing	5
2.2 Malware Analysis	6
2.2.1 Static Malware Analysis	6
2.2.2 Dynamic Malware Analysis	6
2.2.3 Hybrid Malware Analysis	6
2.3 Datasets Modeling Malware Behaviors	7
2.3.1 Static Datasets	7
2.3.2 Dynamic Datasets	7
2.4 Machine Learning	7
2.4.1 Introduction to Machine Learning	8
2.4.2 Machine Learning Pipeline	8

3	Related Work	11
3.1	Methodology	11
3.2	Sandbox Systems	11
3.3	Machine Learning Cybersecurity Solutions	14
4	Requirements and System Design	17
4.1	System Requirements	17
4.1.1	Available Data	17
4.1.2	Algorithm Choices	18
4.1.3	Data Preprocessing	19
4.2	System Design	19
4.2.1	General Architecture	19
4.2.2	Model Trainer Code Section	21
4.2.3	Data Manager Code Section	22
4.2.4	Front-end Charts	22
5	Prototypical Implementation	25
5.1	Initial Test Runs	25
5.2	Implementing the Model Trainer	27
5.2.1	System Call Data	27
5.2.2	Resource Usage Data	29
5.2.3	Model Training Library	29
5.2.4	Trainer Implementation	30
5.2.5	Storing the Trained Models	31
5.3	Extending the Data Manager	32
5.3.1	Classifier Manager	32
5.3.2	Anomaly Detector Manager	34
5.4	Extending the Front-End Interface	34
5.4.1	Model Selector	34
5.4.2	Malware Analysis Tables	35

<i>CONTENTS</i>	xi
6 Prototype Evaluation	37
6.1 Generating the Test Data Set	37
6.2 Model Resource Efficiency	40
6.3 Model Effectiveness	41
6.3.1 Accuracy	43
6.3.2 Precision	43
6.3.3 Recall	44
6.3.4 F1-Score	44
6.3.5 Interpreting the Numerical Results	44
6.4 Model Reproducibility and Explainability	45
7 Summary and Conclusions	47
7.1 Summary of Work Conducted and Conclusion of Main Findings	47
7.1.1 Introduction	47
7.1.2 Background	47
7.1.3 Related Work	48
7.1.4 Design	48
7.1.5 Implementation	49
7.1.6 Evaluation	49
7.2 Further Work	49
7.2.1 Realistic Representation of Sandbox Behavior	49
7.2.2 Changing how the Resource Usage Data is Provided to the Model Trainer	50
7.2.3 Training the Models from the Front-End	50
7.2.4 Integrating the Models' Explainable Metrics into the Front-End In- terface	51
Bibliography	53
Abbreviations	57

Glossary	59
List of Figures	59
List of Tables	61
A Installation Guidelines	65

Chapter 1

Introduction

This chapter describes the motivation behind the project, and outlines the tasks necessary to fulfill the overall goal.

1.1 Motivation

Technology has become a ubiquitous part in the modern world, with a variety of enterprises making an effort to implement technological solutions to the tasks they perform. For example, pharmaceutical companies are attempting to integrate digital technologies into their value chains [1], part manufacturers are exploring the concept of human-robot collaboration in order to improve the efficiency of existing manufacturing processes [2], and machine learning tools have shown a lot of promise in predicting price trends within stock exchange markets [3].

However, this newfound reliance on technology poses great risks, as malicious actors seek to exploit such systems. They are aided by the lack of experience on the side of small and midsize enterprises (SMEs), who are becoming increasingly reliant on such technologies (such as computerized order entry systems in hospitals) [4], yet often lack the resources or do not see the need to invest in advanced cybersecurity solutions to safeguard themselves from attacks [5].

Due to this, there have been noticeable changes in the nature of malware attacks - rather than targeting individual users, many malicious actors now choose to attacks SMEs, for example through ransomwares such as the WannaCry, Erebus, and SamSam viruses, with the economic consequences often exceeding a million dollars per company [6].

Standard antiviruses, which have previously been the answer for protection against such attacks, cannot always detect new forms of malware, which can be the case for variations of malware from the same family, or other, unknown (zero-day) exploits [7]. However, a modern solution to this problem is becoming apparent - machine learning can be applied to detect malware within systems, offering tools for both the detection of known malware

types, as well as the detection of malware making use of previously unknown vulnerabilities [8].

Such tools can safely be implemented into existing enterprise architectures through a sandbox environment. This process is relatively cheap (meaning that it is applicable for small and midsize enterprises), and provides an environment in which files can be ran and analyzed without the risk of damage to the underlying system [9].

Therefore, there exists a need for a cheap, reliable machine learning tool for malware detection within the systems of small and midsize enterprises, which is described by this paper.

1.2 Description of Work

In this thesis, a study is conducted into the mechanisms offered by machine learning for malware detection and analysis within sandbox systems. Existing works within the subjects of "Machine Learning Malware Cybersecurity Solutions" and "Sandbox Systems" are reviewed and critically evaluated, and a design for a machine learning malware analysis sandbox system is proposed, outlining the requirements behind a successful system, and how the system's architecture is implemented.

The proposed design is implemented as a prototype, extending the existing SecBox sandbox system with machine learning pipelines, in particular Decision Tree and Naive Bayes supervised learning models, and Local Outlier Factor and Isolation Forest unsupervised learning models, with the underlying model training mechanism allowing additional models to be trained in a simple, reproducible manner.

The implementation is done in Python, and the trained models are used to generate predictions in real-time with regards to files that the SecBox is executing and monitoring. The models are then evaluated with regards to their usefulness and resource usage in a realistic scenario dealing with a Monti ransomware execution within the SecBox environment.

Finally, the findings of this project are concluded, and possible next steps are proposed, with regards to research areas that are identified as important, and may improve the implemented prototype solution.

1.3 Thesis Outline

Chapter 1 outlines the increasing dependence of SMEs on technology, as well as their lack of investment into existing cybersecurity measures, resulting in increased risk of malware attacks. It establishes the need for a machine learning malware analysis tool, and outlines the structure of this project with regards to the design, implementation and evaluation of this tool.

Chapter 2 looks into the underlying concepts behind a reproducible, machine learning malware analysis system. In particular, sandboxes are discussed with regards to how virtualization techniques allow for safe malware execution without risk to the host machine, malware analysis is discussed in terms of static, dynamic and hybrid malware analysis techniques (and which data they require), and machine learning is discussed in terms of problem types and the machine learning pipelines that are used to train and evaluate models.

In Chapter 3, research is conducted into existing works within the topics of "Machine Learning Cybersecurity Solutions" and "Sandbox Systems". Each cybersecurity solution is critically evaluated in terms of its data format requirements, data processing techniques, machine learning models implemented, as well as metrics such as the solution's accountability and reproducibility of the trained models. Each sandbox system is evaluated in terms of its system requirements, runnable file types, virtualization technique, as well as whether the sandbox is open source, maintained and accountable. SecBox is selected from existing sandbox systems as the base system to expand upon within this project.

Within Chapter 4, a design is proposed for the prototype system. First, available data from the SecBox environment is discussed, and choices are made regarding which data files should be used for the model training. Then, the concrete machine learning algorithms are chosen for both the classification and anomaly detection models, followed by a discussion of the data preprocessing methods to be used. Lastly, the system design is outlined in terms of its existing architecture and proposed changes.

Chapter 5 describes the prototypical implementation of the proposed design, outlining the results of initial malware test runs within the SecBox interface, how the Model Trainer module is implemented with regards to data processing, Scikit-learn model training library and model storage, as well as how the Data Manger module is extended to fulfill the real-time classification and anomaly detection tasks.

In Chapter 6, the implemented solution is evaluated, talking about how the test data set was generated, how efficient each of the implemented machine learning models is in terms of its resource usage, as well as how effective they are based on confusion matrices and derived metrics. The numerical results are interpreted and explained, and the model reproducibility is discussed.

Finally, Chapter 7 summarises the project, discussing the main findings for each section of the project, as well as proposing topics that can be looked into in the future in order to expand upon the project.

Chapter 2

Background

This section discusses the underlying concepts behind a reproducible machine and deep learning malware analysis sandbox system. These can be split into several distinct sections.

2.1 Sandboxing

This section discusses the *sandboxing* technique and its applications in cybersecurity.

A *sandbox* is a technique often used in cybersecurity, where an application can be encapsulated within the underlying host machine [10]. This allows the user to run a potentially unsafe application (e.g. malware) in an environment similar to the host, exploring the application's effects through monitoring and visualization tools, without the risk of causing damage to the host system.

Each sandbox is designed with specific operating systems in mind, and sometimes only supports a limited number of file extensions in order to ensure the security of its environment.

The environment itself is created based on a *virtualization technique* [11]:

1. Full virtualization creates a virtual machine that fully simulates the host. The program can then be run inside this virtual machine on its own operating system. This is fairly slow, but very secure, as there is limited and controlled interaction with the host system.
2. Containerization involves the program being run directly on the operating system of the host computer through a *container engine*, which has the benefit of being much faster compared to full virtualization, but is less secure, as a poorly designed container can easily expose the user's system to malicious actions.

The *SecBox* environment is a good example of a sandbox environment, using containerization to allow the user to run malware on a simulated system, with a malware-free version running alongside. The data from both sandboxes (such as system calls and network activity) is then recorded, providing opportunities to analyze the malware [12].

2.2 Malware Analysis

This section discusses malware analysis, its types and relevance for the proposed solution.

Malware analysis is the study of the functionality, purpose, origin, and potential impact of malicious software. It is typically separated into 3 forms - *Static Malware Analysis*, *Dynamic Malware Analysis* and *Hybrid Malware Analysis*.

2.2.1 Static Malware Analysis

The goal of static malware analysis is to understand a malware sample by studying its source code, which would allow to understand the malware's purpose without needing to execute it on a device [13].

However, in practice, malware samples which can be collected from real systems are only available as binaries - files which contain machine code, which cannot easily be converted back into understandable high-level programming code needed to perform static analysis. Decompiling such code for analysis purposes remains a challenge.

2.2.2 Dynamic Malware Analysis

In dynamic malware analysis, a malware sample is observed at runtime. This is achieved by running it on a system, typically either on a virtual machine or inside a container (which isolates the malware from other applications, but still allows it to access the machine hardware) [14].

For example, the *SecBox* malware analysis sandbox runs a malware sample using container virtualization inside a VM, alongside a non-infected sandbox, and logs resource usages (such as CPU and RAM usage) for both, allowing users to see how various malware samples affect system resources during their runtime.

2.2.3 Hybrid Malware Analysis

Hybrid malware analysis combines the previous two techniques, attempting to combine the advantages and avoid the disadvantages of both. One way it can do so is by executing the malware while performing dynamic analysis, and then performing static analysis on any artifacts that were produced (such as changes in memory made by the malware), allowing

for more useful information to be gathered about the malware sample in question, which may yield better analysis results [15].

Of the three malware analysis methods, Dynamic Malware Analysis is the most relevant type for the project, due to its usage within the *SecBox* environment to study the malware sample's effects at runtime.

2.3 Datasets Modeling Malware Behaviors

This section discusses datasets which can be used to represent malware and record its actions.

In order to train a machine learning model to recognize and classify malware, a dataset which models malware behaviors will be used. To help reproducibility of the machine-learned model obtained from the data set, it must also be preprocessed in a uniform way each time a model is trained (this aspect will be discussed in the next chapter).

Two possible dataset types exist - *Static Datasets* and *Dynamic Datasets*.

2.3.1 Static Datasets

Static datasets are collected through static malware analysis. They typically contain features generated from the malware files (e.g. presence of specific keywords in the code, or file hashes), as well as metadata related to the malware (e.g. file encoding and creation date) [16]. These can be used to identify known malware samples, as well as those which are similar to the samples in the training dataset.

2.3.2 Dynamic Datasets

Dynamic datasets are collected through dynamic malware analysis, and contain data related to how the malware sample behaves at runtime - for example, CPU and RAM usage, network packets sent and received, and system calls (requests from the program for the system's resources or services) among others. Such datasets provide better recognition of heterogeneous malware samples compared to static datasets [17].

2.4 Machine Learning

This section discusses machine learning and the machine learning pipeline.

2.4.1 Introduction to Machine Learning

At its core, *Machine Learning* (ML) is a process of using historical data to create a prediction algorithm for future data, which takes a (training) data set, and outputs a model according to which, new data points can be evaluated.

There exist two main problem types for ML:

- Classification problems, where the goal is to assign an observation (data point) to a specific category.
- Regression problems, where the goal is to predict numerical properties of an observation.

2.4.2 Machine Learning Pipeline

A typical machine learning component is implemented within a system through a *machine learning pipeline*. Such a pipeline which can be outlined in 4 distinct phases [18]:

Data Preprocessing

The raw data needs to be prepared prior to training the model, as the data quality can have major effects on the quality of the model.

First, the data is cleaned, which corrects any errors/invalid data in the data set, such as by removing them altogether, or substituting them with other values (such as the data set average).

Afterwards, *Feature Engineering* is performed, where certain input features are selected or transformed in order to maximize the performance of the model. For example, *Feature Encoding* assigns numeric values to categorical variables. Or, Principal Component Analysis can be used to represent the input data set through a smaller number of variables.

Lastly, *Data Splitting* is performed, where the data set is split into a *training set*, which will be used to train the model, and a *test set*, which will be used to evaluate the model performance. *Conventional validation* consists of a 70-30 split between the training and test data, whereas *cross-validation* requires the model to be trained with multiple possible splits of the data, which performs better for small data sets, where a split can hide statistical properties of the data.

Model Training

The preprocessed data is given to a selected machine learning algorithm, which attempts to learn patterns from the data set. A multitude of ML algorithms exist [19]:

Supervised learning algorithms are trained using a labeled data set, and use probabilities of previously observed events to infer probabilities for new data.

Unsupervised learning algorithms are given unlabeled data, try to infer patterns from it, and apply these to new data.

Model Validation

To test the prediction accuracy of the model, its performance on the test data set is evaluated, measuring metrics such as:

1. Precision: $\text{True Positives} / (\text{True Positives} + \text{False Positives})$
2. Recall: $\text{True Positives} / (\text{True Positives} + \text{False Negatives})$
3. F1 score: $2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$

This stage helps identify problems with the model and how it was trained, e.g. *overfitting*, where the model memorizes the training data rather than looking for patterns within, which yields a high prediction accuracy for the training data, but very low accuracy for any other data set [20].

In the case of poor model performance, the pipeline may need to be revised, such as changing how data was preprocessed, or selecting a different ML model to be trained.

Model Deployment

Once the model has been trained and demonstrated satisfactory performance during validation, it is deployed into its intended system, where it will make predictions about new data.

Chapter 3

Related Work

This chapter looks at real-world systems implementing sandbox environments and machine learning pipelines.

3.1 Methodology

This section details the methodology according to which related work will be evaluated. Two types of solutions will be looked at:

1. Sandbox systems for malware analysis
2. Machine learning cybersecurity solutions

Each sandbox system will be evaluated in terms of its platform compatibility, handled data types, virtualization method and general characteristics.

Each ML cybersecurity solution will be evaluated in terms of its ML pipeline (see 2.4.2), scope of application, and limitations compared to the proposed solution.

3.2 Sandbox Systems

In addition to the previously mentioned *SecBox* environment, other sandbox systems with relevance for malware analysis exist. This section details several such systems.

Kaspersky Sandbox [21] implements a full virtualization solution for dynamic malware analysis, in which it receives specific requests regarding the file to execute, environment and file configuration, and operating system.

The file is executed, with the sandbox monitoring its interactions with the OS through various means. Based on the analysis result, it labels the file as either malware or benign, while including providing activity logs from the execution process. It also implements VM

environment randomization and simulated user actions, preventing VM detection by the malware. The sandbox is able to handle all Windows files, in addition to Android APKs and internet links.

Unfortunately, the solution is patented, closed-source and does not provide real-time monitoring of the file execution.

Open-source solutions do exist, for example Cuckoo Sandbox [22] is a VM sandbox for automated analysis of suspicious files. It provides high extensibility and customization thanks to its modular architecture and open-source model. It is available for most major operating systems, and consists of a central managing software (the Host) and multiple possible virtual environments (Guests)

To submit a file for analysis, users can use the Submission Utility, with the option to either use the default analysis packages provided, or add their own. Then, the file will be executed with the environment options specified, and several results files will be generated and saved by the sandbox (such as Cuckoo process monitoring logs).

However, the current version of Cuckoo is unmaintained, does not implement machine learning algorithms by default, and lacks real-time monitoring capacity.

In contrast to these sandboxes, not every sandbox comes with a software suite to facilitate malware analysis functions - DRAKVUF [23] is a black-box binary analysis system for dynamic malware analysis, designed for Windows and Linux systems (which must run on Intel CPUs with virtualization support). It supports all file types, and is open source.

The aspect that sets it apart from other malware analysis sandboxes is its agentless approach - instead of needing additional software, it takes advantage of existing tools in the operating system, using techniques such as process injection to run files on the host, for example by hijacking the Windows Task Manager execution.

But, DRAKVUF doesn't implement machine learning, making it vulnerable to novel malware threats. It also can't be executed on CPUs other than Intel due to its reliance on the Intel virtualization support. Lastly, the only real-time monitoring function is a console log, which can be hard to interpret.

Some sandboxes, such as Limon [24], implement machine learning tools. It is a Linux sandbox, which uses full virtualization for malware analysis. It allows the user to perform both static analysis prior to malware execution (e.g. md5 hash, ELF header), dynamic analysis with real-time metrics as the software is being executed (e.g. syscalls, packet captures), as well as post-mortem analysis after the VM is suspended.

Limon is open-source, and supports ELF executables, Loadable Kernel Modules (LHM) as well as Python, Perl, Shell, Bash and PHP scripts. Unfortunately, it is not maintained, and implements black-box machine learning algorithms through the VirusTotal API. Also, since it is only available for Linux machines, it's less versatile compared to malware analysis tools that supports multiple platforms, since it cannot be used to detect common threats for other platforms.

Finally, Any.Run [25] is a cloud-based tool for dynamic malware analysis. The user can submit files for analysis, interact with the sandbox environment in real-time, as well as use the internet to check websites for web threats. The results are presented in an "ATT&CK Matrix", providing visual feedback and accountability with regards to how the file was classified.

However, the sandbox is closed-source, and does not specify which (if any) machine learning algorithms it implements, functioning like a black box.

The comparison table below uses the following metrics:

1. Platform(s) - which operating system(s) is the sandbox designed for?
2. Artifact Types - which kinds of files can the sandbox run?
3. Virtualization Technique - Does the sandbox use full virtualization or containerization?
4. Open Source - Is the sandbox open-source?
5. Maintained - Is the sandbox actively maintained?
6. Accountable - Does the sandbox implement tools to track its actions and the analysis progress (e.g. real-time monitoring)?
7. Implements ML - Does the sandbox implement any machine learning algorithms for malware analysis?

Table 3.1: Comparison of Sandbox Systems

System	Platform(s)	Artifact Types	Virtualization Technique	Open Source	Maintained	Accountable	Implements ML
Kaspersky Sandbox (2019)	Windows, Android	All Windows Files, Android APKs	Full Virtualization	No	Yes	No	Yes (undisclosed algorithms)
Cuckoo Sandbox (2011)	Windows, Linux, Mac OS, Android	Most Windows files, Android APKs, Additional files as defined by the user	Full Virtualization	Yes	No	Yes	No
DRAKVUF (2022)	Windows, Linux (only Intel CPUs)	All files	Full virtualization	Yes	Yes	No	No
Limon (2015)	Linux	ELF, LKM, Programming language scripts	Full Virtualization	Yes	No	Yes	Yes (undisclosed algorithms)
Any.Run (2023)	Windows	Most Windows files	Full virtualization	No	Yes	Yes	Yes (undisclosed algorithms)
SecBox (2023)	Linux	Bash scripts, ELF files	Containerisation	Yes	Yes	Yes	No

Overall, it can be seen that existing sandbox solutions are either closed-source (lacking accountability with regards to why certain decisions were made regarding the data), unmaintained, or lack machine learning components (rendering them ineffective against unseen malware threats).

However, we can also note that the *SecBox* system meets most of the chosen criteria for a malware analysis system. It only lacks a machine learning component, which makes it a perfect candidate as the system to be extended for this project.

3.3 Machine Learning Cybersecurity Solutions

This section reviews existing machine learning solutions that have been developed for cybersecurity and malware analysis.

In [26], the authors implemented a stacked deep learning approach, performing basic data preprocessing by removing missing values and standardizing numeric features, then training 5 deep learning models with the training data, and taking the model averages in order to generate predictions.

The constructed model was evaluated using a SCADA dataset from the Mississippi State University, with data present for specific attack types, and could then be used for intrusion detection within SCADA systems.

Unfortunately, it was developed exclusively for SCADA-specific threats and scarce data, and is not able to take advantage of larger data sets if those are available.

An example of a system that can analyze data from operating systems is [27], a framework for automatic, incremental analysis of malware samples.

The solution uses system calls log files collected during malware execution in a sandbox environment, preprocessing the data by mapping each system call to a vector in a multi-dimensional vector space.

Each dimension of this specially designed vector space represents a certain behavioral pattern, allowing clustering to be applied in order to distinguish data categories, followed by classification in order to assign new data samples to the known categories. The framework's performance is evaluated using metrics such as Precision, Recall and F-Score.

However, the framework only provides support for analysis using the specific data mapping and representation procedure described, meaning it is not applicable for implementing other data preprocessing steps or training different ML models.

System calls are not the only possible aspect for machine learning - [28] implements multiple algorithms for malware classification based on the PE file headers. The solution takes a preprocessed data set of PE header attributes from Kaggle, where each row of the dataset represents an executable file, and is split into 57 columns, each representing a PE header feature. Several ML methods are then applied to the dataset, such as Decision Trees and Gradient Boosting, each training a model. The performance of all trained models is then compared based on their accuracy scores. But, the proposed approach only works with preprocessed PE header data, and fails to implement the preprocessing part of the ML pipeline, as well as lacking support for other malware analysis metrics (e.g. syscalls).

A more focused approach was taken in [29], where a malware sample dataset from the Canadian Institute of Cybersecurity was taken. The dataset consists of memory dumps from both benign and malicious memory dumps, and was preprocessed using feature selection and feature rank to reduce its dimensionality, allowing to train ML models on the important features of the data

Several ML classification algorithms (such as KNN, SVM) were then trained on the data. Once the models were trained, they were evaluated based on their accuracy scores in classifying malware samples from a test set. The model that showed the best accuracy would be selected. Unfortunately, the paper only includes support for the specific malware

data set provided, which quickly becomes obsolete in a world of ever-evolving malware threats.

Finally, in [30], an ontology-based machine learning approach was applied, where an Android application ecosystem was represented formally as an ontology based on a method described in the paper, and a *Bag of Graphs* technique was used to find common properties of different applications based on their application manifest XML files.

Later, feature importance analysis was performed to select the most important dimensions of the generated dataset, which were then provided to a Random Forest model for training. The model's performance was evaluated using statistical techniques such as Precision, Recall and F1-score.

However, the solution only analyzed Android application manifest XML files, therefore lacking support for dynamic data sets and other data formats, which is important to counter modern malware obfuscation techniques.

The comparison table below has the following metrics:

1. Data format - what is the data format of the training dataset(s)?
2. Data preprocessing - what techniques (if any) were used to preprocess the data prior to model training?
3. Flexible preprocessing - can choices be made regarding how data is preprocessed?
4. ML algorithm(s) - which machine learning algorithm(s) does the solution implement?
5. Accountable - does the solution provide explanations for its predictions, allowing users to understand why a particular decision was made?
6. Reproducible - can a given result (e.g. malware classification) be reliably reproduced when starting with a fresh ML model?
7. Adaptable - can a different ML pipeline be integrated into the solution?

Table 3.2: Comparison of Machine Learning Cybersecurity Solutions

Solution	Data Format	Data Pre-processing	Flexible Preprocessing	ML Algorithm(s)	Accountable	Reproducible	Adaptable
[26] (2022)	MODBUS	Removing missing values, Standardizing numeric features	No	Neural Networks	No	Yes	No
[27] (2011)	System call log files	Mapping function to a vector space	No	Hierarchical Clustering, Nearest Prototype Classification	Yes	No	No
[28] (2022)	CSV	N/A	No	Decision Tree, Gradient Boosting	Yes	No	No
[29] (2022)	Memory dumps	Feature Selection, feature rank	No	KNN, CNN, Naive Bayes, Random Forest, SVM, DT	No	No	Yes
[30] (2018)	Application Manifest XMLs	Ontology model, Bag of Graphs	No	Random Forest	Yes	No	Yes

Overall, it can be seen that existing solutions for malware analysis using machine learning do not implement flexible data preprocessing, either lack reproducibility or accountability with regards to their analysis results, as well as lacking support for implementation of additional machine learning pipelines.

Chapter 4

Requirements and System Design

In order to implement a machine learning tool for a malware analysis sandbox, the tool's requirements must be specified, and an appropriate system design needs to be suggested. This chapter discusses these two aspects.

4.1 System Requirements

4.1.1 Available Data

The *SecBox* environment provides multiple types of system logs that describe a program's activity during its execution. Of particular interest are three of these:

1. System calls CSV files - store all system calls with their timestamp, arguments, as well as additional data.
2. Network activity PCAP files - store network packet captures, with details such as protocol (TCP or UDP) and destination IP addresses.
3. Resource usage JSON files - store information about how the system is performing in real-time, such as percentage CPU usage, RAM usage as well as amount of sent and received packages

Within this thesis, the system call and resource usage files will be focused on, because PCAP files are more difficult to interpret, and may as a result lead to worse model performance when trying to detect malware inside the sandbox.

Both of the chosen file categories are generated by the sandbox for the healthy and infected sandbox instances, and can be used to train machine learning models. However, such data cannot be used out-of-the-box by ML algorithms - different algorithms require different data formats and preprocessing steps to be undertaken prior to model training, so the ML algorithms need to be chosen first, followed by choosing the data preprocessing techniques that will be used.

4.1.2 Algorithm Choices

The machine learning algorithm choices are very important, because one of the key goals for the tool is its accountability - this means that a decision made by an algorithm (e.g. malware/not malware in a simplistic case) should be explainable to a potential user with regards to why this decision was made.

Another goal is reproducibility, which means that it should be possible to take an un-trained machine learning model, and, after training it on the same data set in the same manner, it should generate the same predictions about the data.

Lastly, it's important for the system to be able to recognize both common threats (such as known ransomware, e.g. Monti) and novel threats.

With these 3 criteria in mind, the proposed solution consists of 2 parts, with classification algorithms used for detecting known malware classes, and anomaly detection algorithms for detecting novel threats:

For the **classification** algorithms, the following models will be used:

1. Decision Tree Algorithm
2. Naive Bayes Algorithm

The Decision Tree algorithm was chosen because it has good accountability - this is due to its tree-like structure and feature importance rankings, which show the features of the data that were looked at in order to make the classification decision.

Likewise, Naive Bayes also has a good degree of accountability, with the possibility of determining the feature importance rankings based on the conditional probabilities calculated.

Both of these algorithms are reproducible in terms of their standard code implementations by controlling the random seed parameter.

For the **anomaly detection** algorithms, the following models will be used:

1. Isolation Forest Algorithm
2. Local Outlier Factor Algorithm

The Local Outlier Factor (LOF) algorithm is accountable thanks to its anomaly score, which explains how much of an outlier a data point is when compared to other points.

Similarly, the Isolation Forest Algorithm also provides accountability in the form of scores, which indicate how isolated each data point is.

Both of these algorithms can be used in a reproducible manner by controlling the random seed, as well as the **n_neighbors** and **contamination** parameters.

4.1.3 Data Preprocessing

All of the aforementioned algorithms require data to be preprocessed in order to function effectively. Therefore, the available system call CSV files and resource usage JSON files need to be converted to a suitable format.

System Call Files

For the system call files, a bag-of-words algorithm was selected for the feature extraction, with two data preprocessing options:

Option 1 is frequency-based preprocessing: The system calls present in the file will be collected together into time slices, thus accounting for differences in execution time. For each time slice, a 2D array will be generated, with time slices as the rows, and counters for each system call and how often it appears within the time slice as the columns. For an illustration, see Table 5.1.

Option 2 is sequence-based preprocessing: Instead of counting how often each system call appears in a time slice, the algorithm will instead count how often every other system call appears after the system call currently being considered (within each time slice). For an illustration, see Table 5.2.

In both cases, the resulting array can be used to train the system call machine learning models.

Resource Usage Files

For the resource usage files, the individual metrics (CPU percent usage, RAM usage, packages sent, packages received) will be combined together into a 2D array. For a visualisation of the resulting data format, see Table 5.3.

Note that, unlike the system call data, these operations already yield a data format which can be used to train the resource usage machine learning models, so there is no need to perform further data processing such as splitting data into time slices (unless the processing method proves to be ineffective in terms of generating good models).

4.2 System Design

4.2.1 General Architecture

The existing SecBox architecture is shown in Figure 4.1.

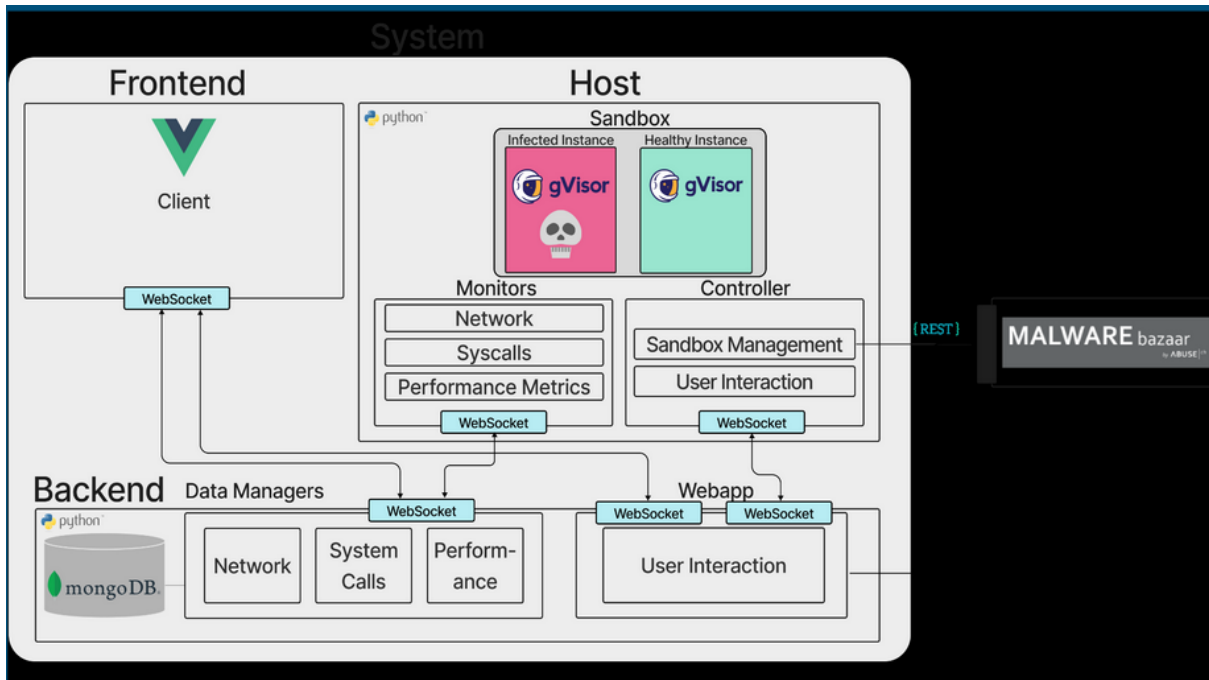


Figure 4.1: SecBox existing architecture

As it can be seen, there are 3 main components - Frontend, Host and Backend. Each can be run separately on different machines if needed, and all are connected together via WebSockets. There is also a REST API connection to Malware Bazaar, which is used to access new malware samples.

The host is responsible for setting up and running the sandbox on a target machine, making use of gVisor to provide isolation between the sandbox and the hardware components. It includes several Monitors to record data regarding the sandbox runs:

- Network Monitor records data such as IP addresses that the sandbox sends packets to and the transmission protocol (TCP or UDP). Since the PCAP files are not relevant for this project, this monitor can be ignored.
- Syscall Monitor records each system call performed by the system, with data such as timestamp, system name, thread id and additional arguments.
- Performance Monitor records the resource usage of the system at every point in time, with metrics such as CPU % usage, RAM usage, and packages sent and received.

The backend is responsible for receiving the data sent by the host monitors, performing data processing for visualizing the data within graphs in the frontend, as well as accepting user instructions for the sandbox environment from the frontend and passing them to the host controller.

The frontend provides a graphical user interface for launching and managing the sandbox environment to the user, visualizes data from the sandbox execution in real-time, as well as showing the results of previous sandbox runs in a post-mortem analysis tab.

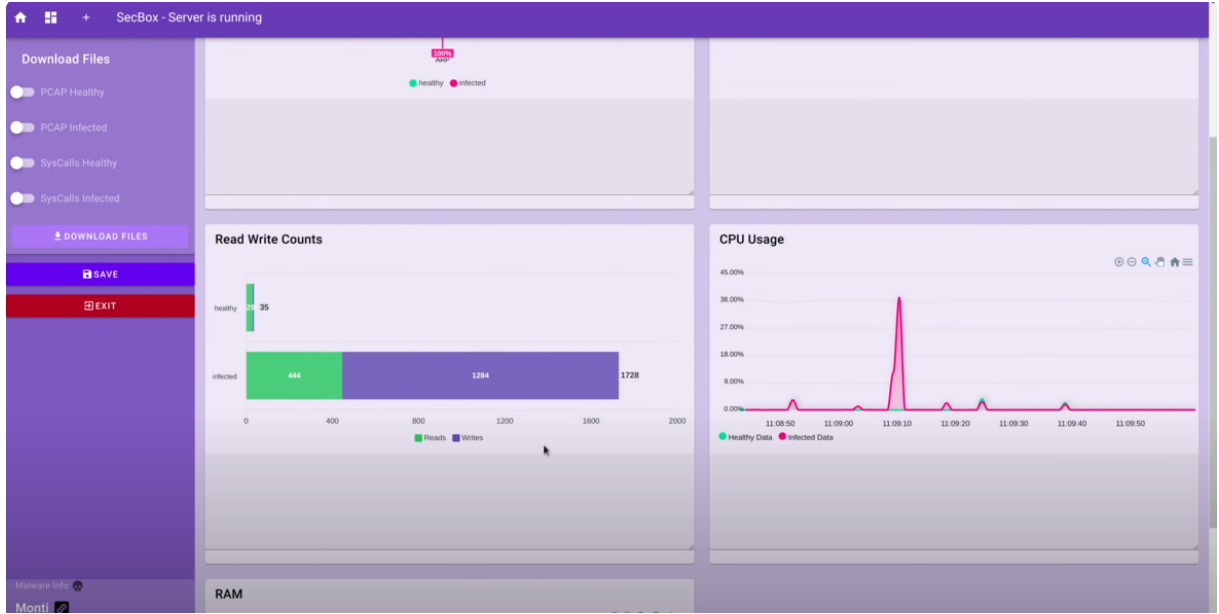


Figure 4.2: SecBox post-mortem analysis tab

The proposed extension of the SecBox consists of several parts, discussed below.

4.2.2 Model Trainer Code Section

First, a **ModelTrainer** module will be added to the backend. It will contain the data needed to train the ML models (syscall and resource usage files), an abstract **model-Trainer.py** class, and the concrete implementations for the different types of machine learning models used in the project:

- Resource Usage Classifier
- Resource Usage Anomaly Detector
- System Call Classifier
- System Call Anomaly Detector

Each of these will implement the training pipelines discussed above, as well as a "predictor" function, which will take a trained model as well as a dataset, and make a prediction regarding whether the dataset comes from a malware-infected system or not. The trained models will be stored as part of the module, providing easy access to them again. Each model will be linked to a configuration file, which will record training parameters such as dataset used, random seed, etc. in order to ensure reproducibility of the model training.

4.2.3 Data Manager Code Section

To link the Model Trainers to the Front-End, the existing Data Manager Module (which is currently responsible for processing and delivering performance and system call data to the front-end) will be extended with two new Data Managers, both of which will integrate into the existing WebSocket connections:

1. **anomalyDetectorManager.py** will be responsible for accepting data from the front-end in real-time, converting it into an acceptable format for generating a prediction, as well as instructing the corresponding (system call or resource usage, depending on the data received) Anomaly Detector component to load the appropriate model and generate a prediction based on the data. This prediction will then be forwarded to the Anomaly Detector Table in the front-end.
2. **classifierManager.py** will also be responsible for accepting data from the front-end in real-time, preprocessing it and instructing the corresponding Classifier component to load an appropriate model and generate a prediction, which will be forwarded to the Malware Analyzer Table in the front-end.

4.2.4 Front-end Charts

Apexcharts is currently used as the graphing library for the front-end interface. The existing real-time analysis page (see Figure 4.3) will be extended with two additional tables mentioned in the previous section:

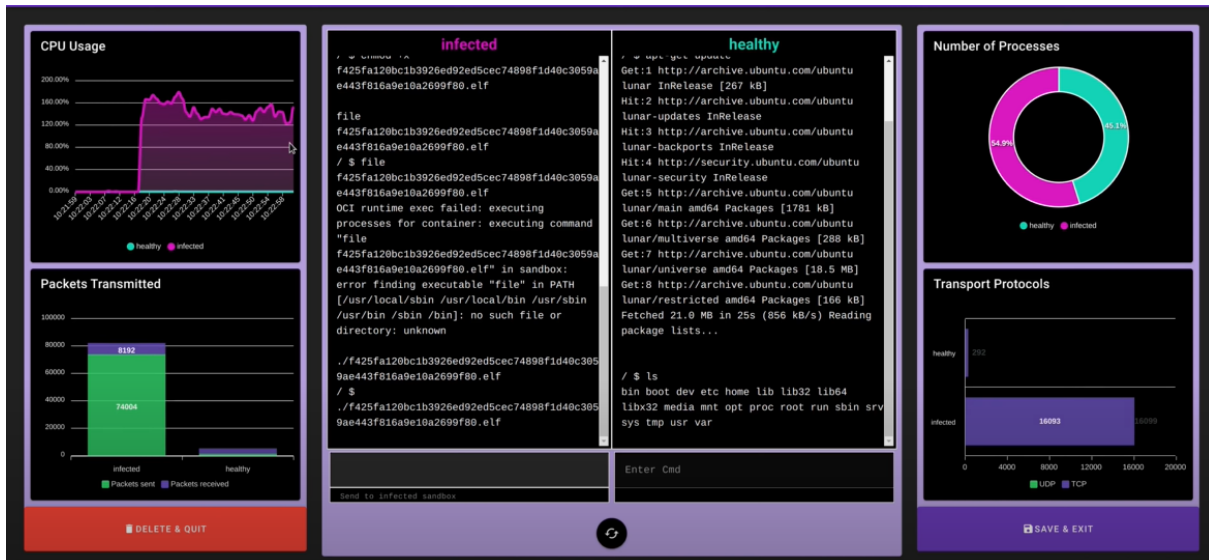


Figure 4.3: SecBox existing analysis page

- The Malware Analyzer Table will display the time slices processed by the models, and the corresponding predictions from both the System Call Classifier component and Resource Usage Classifier component. Such predictions may support both

binary (infected/healthy) and multiclass (Monti, CoinMiner, healthy, etc.) classifications, depending on how the data sets are labelled.

- The Anomaly Detector Table will display the time slices processed by the models, and the corresponding predictions from the System Call Anomaly Detector and Resource Usage Anomaly Detector components. Such a prediction with either be "inlier" or "outlier".

Chapter 5

Prototypical Implementation

This chapter discusses the concrete implementation of the proposed solution in Chapter 4.

5.1 Initial Test Runs

After the design choices were finalized, actual work with the **SecBox** environment began. For the initial setup, it was necessary to create .env files for each individual section of the SecBox (front-end, back-end and host) in order to specify how they should run together, as each one could also be run on a separate machine. An example of a configured .env file can be seen in Figure 5.1

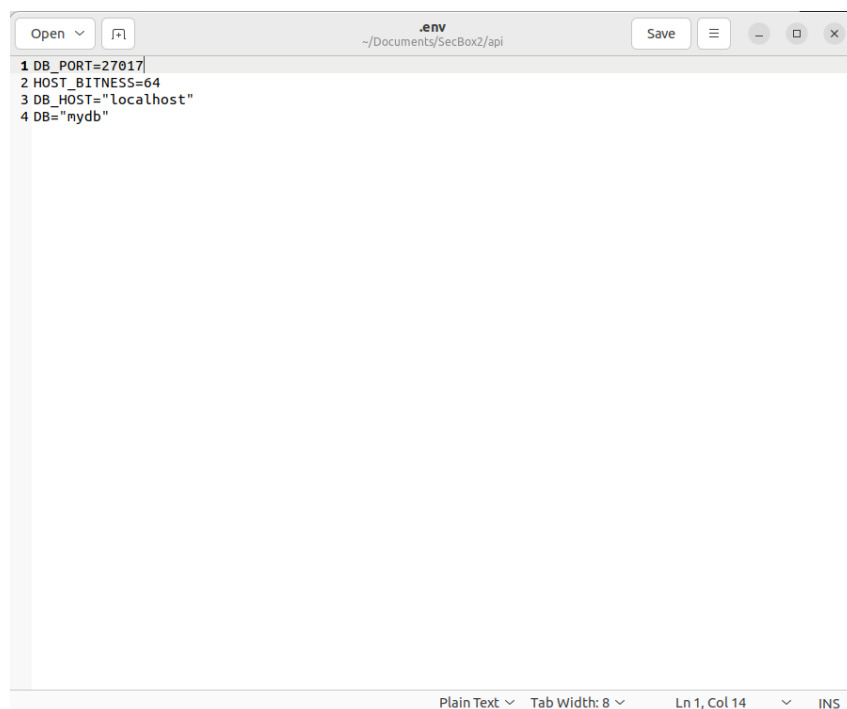


Figure 5.1: A configured .env file for the SecBox api

Afterwards, several sandbox test runs were conducted, running a random available malware on different available operating systems. It was determined that

1. Certain malware types did not work on the sandbox (e.g. the provided CoinMiner malware sample), see Figure 5.2.
2. Other malware types (e.g. Monti ransomware) consistently worked well.

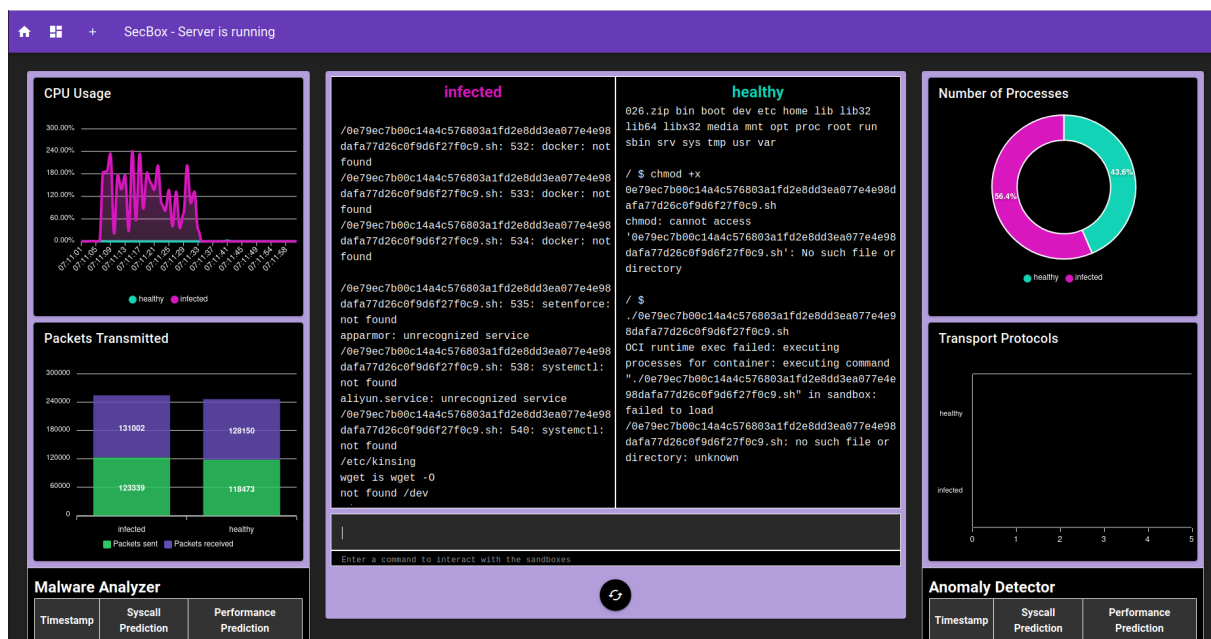


Figure 5.2: SecBox error when attempting to run a CoinMiner virus - various libraries not found

As a result, the requirements were adjusted to have a specific focus on the Monti malware family, which is a ransomware that encodes system files.



Figure 5.3: Monti ransomware in action within SecBox - note the CPU usage spike and encrypted disk files

There were several reasons for this decision:

1. It was shown to execute reliably within the SecBox environment.
2. It should be more easily detectable using the chosen machine learning algorithms thanks to its aggressive resource usage during the disk encryption phase.

The aforementioned properties will also make it easier to prove that the machine learning models function correctly, which is great for the proof-of-concept prototypical implementation.

5.2 Implementing the Model Trainer

5.2.1 System Call Data

The first stage of implementing the ML Model Trainer was the data preprocessing. For the system call data, two pipelines were implemented (as discussed in the design section):

1. Frequency-Based Preprocessing
2. Sequence-Based Preprocessing

For both, the initial system call file (see Figure 5.4) was reduced to just the columns containing the important information (in this case, they are **time_ns** and **sysname**)

	A	B	C	D	E	F	G	H	I	J	K	L	M	
1	time_ns	thread_id	syscall	system	container_id	code	credentials	args						
2	1.6868163689292E+018	1	12	hrk	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		[0	72	77	0	22	13925620083530	
3	1.68681636893163E+018	1	63	uname	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139318974356824	139256203116776	139256202965360	15032384000	49	15032381856]	
4	1.68681636893245E+018	1	21	access	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256200833794	0	1	15032384000	0	139256203116776]	
5	1.68681636893318E+018	1	21	access	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256200845440	4	139256200921112	8	139256202968147	1]	
6	1.68681636893344E+018	1	257	openat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(4294967196	139256200834728	524288	0	65535	0]	
7	1.68681636893412E+018	1	5	fsstat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	139318974354256	139318974354256	0	65535	0]	
8	1.68681636893457E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(0	7534	1	2	3	0]	
9	1.6868163689347E+018	1	3	close	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	7534	1	2	3	0]	
10	1.68681636893519E+018	1	21	access	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256200833794	0	0	2	1	0]	
11	1.68681636893575E+018	1	257	openat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(4294967196	139256202968528	524288	0	0	13931897435468	
12	1.68681636893607E+018	1	0	read	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	139318974354712	832	0	0	13931897435468	
13	1.6868163689362E+018	1	5	fsstat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	139318974354352	139318974354352	139256202968528	139256202965360	139256202965360	
14	1.68681636893634E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(0	8192	3	34	4294967295	0]	
15	1.68681636893646E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(0	2267936	5	2050	3	0]	
16	1.68681636893656E+018	1	10	protect	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256196665056	2093056	0	139318974353872	3	0]	
17	1.68681636893662E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256198746112	20480	3	2066	3	151552]	
18	1.68681636893683E+018	1	3	close	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	40	0	49	1879048191	1879048225]	
19	1.6868163689369E+018	1	21	access	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256200833794	0	0	139256200813094	1	0]	
20	1.68681636893951E+018	1	257	openat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(4294967196	139256203097296	524288	0	0	13931897435463	
21	1.68681636893981E+018	1	0	read	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	139318974354664	832	0	0	13931897435463	
22	1.68681636893993E+018	1	5	fsstat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	139318974354304	139318974354304	139256203097296	139256203097296	139256202965360	
23	1.68681636893997E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(0	2109712	5	2050	3	0]	
24	1.68681636894005E+018	1	10	protect	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256192315392	2093056	0	139318974353824	3	0]	
25	1.68681636894009E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256194408448	8192	3	2066	3	8192]	
26	1.68681636894054E+018	1	3	close	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	40	0	49	1879048191	1879048225]	
27	1.686816368941E+018	1	21	access	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256200833794	0	0	139256200814064	1	0]	
28	1.6868163689411E+018	1	257	openat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(4294967196	139256203098576	524288	0	0	13931897435459	
29	1.68681636894146E+018	1	0	read	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	139318974354616	832	0	0	13931897435459	
30	1.68681636894156E+018	1	5	fsstat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	139318974354256	139318974354256	139256203098576	139256202965360	139256202965360	
31	1.68681636894159E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(0	411552	5	2050	3	0]	
32	1.68681636894166E+018	1	10	protect	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256190103552	2097152	0	139318974353632	3	0]	
33	1.6868163689417E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256192200704	24576	3	2066	3	1994752]	
34	1.68681636894227E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256192252260	15072	3	50	4294967295	0]	
35	1.68681636894245E+018	1	3	close	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(3	40	0	49	1879048191	1879048225]	
36	1.68681636894285E+018	1	9	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(0	12288	3	34	4294967295	0]	
37	1.68681636894318E+018	1	158	arch_procl	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(4088	139256203086532	139256203086016	3	34	4294967295	0]
38	1.68681636894346E+018	1	10	protect	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256192200704	16384	1	13925620309608	139256203116940	6]	
39	1.6868163689439E+018	1	10	protect	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256194408448	4096	1	139256203097344	1	1947]	
40	1.68681636894437E+018	1	10	protect	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256198746112	16384	1	139256203096064	0	2007]	
41	1.68681636894503E+018	1	10	protect	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(9489751511552	16384	1	139256202965360	0	1127]	
42	1.68681636894533E+018	1	10	protect	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256202956800	4096	1	139256202963440	0	193]	
43	1.68681636894558E+018	1	11	mmap	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(139256203104256	7534	4110283702272	139256202963440	0	193]	
44	1.68681636894646E+018	1	257	openat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(4294967196	94897549211903	2050	0	0	139256192224640	139256192224640
45	1.68681636894662E+018	1	5	fsstat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(0	21955	139318974356000	139256192224640	139256192224640	139256192224640	
46	1.68681636894707E+018	1	5	fsstat	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(0	139318974356064	139318974356064	0	139318974356000	139256192224640	
47	1.68681636894751E+018	1	12	hrk	5b0d00c9944bc3f3d5852a6bfe4e7b5787d14c0d3f982d06130c328d8045c	/		(0	139256192220224	131696	1.84467440737095E+019	2	0]	

Figure 5.4: Example system call CSV file

Then, for the frequency preprocessing procedure, the system calls were collected into time slices of 1 second - this time period was chosen because it had the best tradeoffs in terms of including system calls that took a long time to execute (if a system call takes longer than the specified time period, it will lead to empty time slices when preprocessed, cluttering the data), and not having vastly unbalanced time slices, where some would have in excess of 100 system calls, while others would only have 1.

Each time slice would thus have a counter for how often each system calls appeared inside, leading to the data format shown in Table 5.1 (note that only a few system calls are shown for clarity, rather than a complete list). This format matches the proposed frequency data structure in the design section of this report, and was used to train the system call classifier and anomaly detector models.

Table 5.1: Sample frequency-processed system call list

Timestamp	readlink	mkdir	unlinkat	rt_sigaction	lstat
1679827200000	5	0	1	0	1
1679827201000	0	1	0	6	0
1679827202000	1	0	2	0	1
1679827203000	0	1	0	1	0
1679827204000	1	0	1	0	1
1679827205000	0	9	0	12	0
1679827206000	1	0	1	0	1
1679827207000	0	55	0	1	0
1679827208000	1	0	18	0	1
1679827209000	4	13	2	1	0

If sequence preprocessing was chosen instead, each entry would include a list rather than a counter, with each entry in that list corresponding to how often a system call in the list appeared after the current system call being considered, within a particular time slice. For a visualization of this data structure, see Table 5.2.

Table 5.2: Sample sequence-processed system call list

Timestamp	readlink	mkdir	unlinkat	rt_sigaction	lstat
1679827200000	[1,0,0,1,1]	[0,1,1,0,1]	[1,0,1,1,0]	[0,1,0,1,1]	[1,0,0,1,1]
1679827201000	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]
1679827202000	[1,0,0,1,1]	[0,1,1,0,1]	[1,0,1,1,0]	[0,1,0,1,1]	[1,0,0,1,1]
1679827203000	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]
1679827204000	[1,0,0,1,1]	[0,1,1,0,1]	[1,0,1,1,0]	[0,1,0,1,1]	[1,0,0,1,1]
1679827205000	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]
1679827206000	[1,0,0,1,1]	[0,1,1,0,1]	[1,0,1,1,0]	[0,1,0,1,1]	[1,0,0,1,1]
1679827207000	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]
1679827208000	[1,0,0,1,1]	[0,1,1,0,1]	[1,0,1,1,0]	[0,1,0,1,1]	[1,0,0,1,1]
1679827209000	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]	[1,0,1,0,1]	[0,1,0,1,0]

Likewise, we can see that the sequence data structure matches the proposed structure in the design section.

5.2.2 Resource Usage Data

For the resource usage data, the important outlined resources (cpu percent usage, ram usage, packages sent, packages received) were collected into the following format:

Table 5.3: Resource usage processed data

Timestamp	CPU %	RAM Usage	Packages Sent	Packages Received
1679827200000	0.75	1215	32	19
1679827201000	0.35	836	12	8
1679827202000	0.92	562	8	15
1679827203000	0.23	938	23	37
1679827204000	0.68	457	15	12
1679827205000	0.42	1043	7	22
1679827206000	0.81	274	20	9
1679827207000	0.17	1599	32	30

5.2.3 Model Training Library

The concrete machine learning model implementations were taken from SciKit-learn, as it is a popular python library for machine learning, providing all the necessary tools for implementing the proposed solution. The specific models that were used are:

- `sklearn.tree.DecisionTreeClassifier`
- `sklearn.naive_bayes.GaussianNB`
- `sklearn.ensemble.IsolationForest`
- `sklearn.neighbors.LocalOutlierFactor`

5.2.4 Trainer Implementation

The model trainers are implemented as shown below. Note that only some of the code is shown for brevity:

System Call Classifier

Listing 5.1: System Call Classifier

```
class SyscallClassifier(ModelTrainer):
    def __init__(self, model):
        super().__init__()
        self.model = model
        self.all_syscalls = [#list of system calls to be considered]

    def predict(self, data):
        return self.model.predict(data)

    def trainModel(self, dataFilePaths, featureExtractor, modelName):
        self.dataFilePaths = dataFilePaths
        self.featureExtractor = featureExtractor
        self.modelName = modelName

        features_combined_list = []
        marker_combined_list = []
        for path, marker in dataFilePaths:
            syscall_raw_list = self.readFileContents(path)
            features = self.extractFeatures(syscall_raw_list)
            features_combined_list.extend(features)
            marker_combined_list.extend(
                [marker for _ in range(len(features))])

        self.fitModel(features_combined_list, marker_combined_list)
        self.saveModel(modelName)
```

The system call classifier class fulfills two purposes:

1. Allow for models to be trained by initializing the class with the desired Scikit-learn model, then using the **trainModel()** function together with the data file paths, and a feature extraction method (either frequency or sequence-based extraction). The model is then saved as a pickle file.
2. Allow for prediction to be generated in real-time using existing models, by allowing pre-trained models to be loaded by initializing the class with the pretrained Scikit learn model, then generating predictions for data points with the help of the **predict()** function.

Resource Usage Anomaly Detector

Listing 5.2: Resource Usage Anomaly Detector

```
class PerformanceAnomalyDetector(ModelTrainer):
    def __init__(self, model):
        super().__init__()
        self.model = model

    def predict(self, data):
        return self.model.predict(data)

    def trainModel(self, dataFilePaths, modelName):
        self.dataFilePaths = dataFilePaths
        self.modelName = modelName

        features_combined_list = []
        for path in dataFilePaths:
            features = self.readFileContents(path, 'healthy')
            features_combined_list.extend(features)
            features = self.readFileContents(path, 'infected')
            features_combined_list.extend(features)

        self.fitModel(features_combined_list)
        self.saveModel(modelName)
```

The Anomaly Detector classes fulfill the same general purpose (described above) as the Classifier classes, with the difference that no labels are provided during the model training stage. Also, models making use of resource usage files do not require a feature extractor to be specified (unlike the system call models), since the existing features are simply aggregated before being passed to the model.

5.2.5 Storing the Trained Models

Once the data is preprocessed and the model is trained, it is stored as part of the Model Trainer module (see Figure 5.5) using the "pickle" python object serializer. This means

that the models can then be easily loaded back into the trainer class on demand, and be used to generate predictions about new data without having to retrain the model.

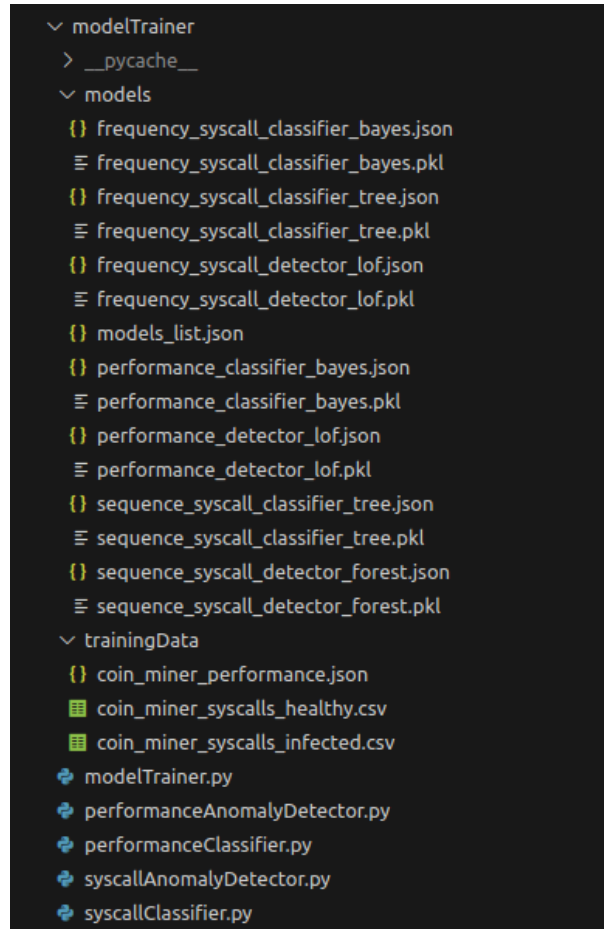


Figure 5.5: Sample stored models as part of the modelTrainer module

5.3 Extending the Data Manager

The data manager module is extended with two further data managers to handle the exchange of data and commands related to the classifiers and anomaly detectors between the front-end interface, back-end model trainers, and host monitors.

5.3.1 Classifier Manager

Listing 5.3: Classifier Manager

```

class ClassifierManager(DataManager):
    def __init__(self, socketio, db,
                 syscall_classifier, performance_classifier):
        super().__init__(socketio, db)
        cwd = os.getcwd()

```

```

with open(#models list) as json_file:
    models = json.load(json_file)

if syscall_classifier in models["categories"]:
    matching_model = models["categories"][syscall_classifier][0]
    with open(os.path.abspath(matching_model), 'rb') as file:
        trained_model = pickle.load(file)
    self.syscall_classifier = SyscallClassifier(trained_model)
    if 'frequency' in syscall_classifier:
        self.syscall_classifier.featureExtractor = 'frequency'
    else:
        self.syscall_classifier.featureExtractor = 'sequence'

if performance_classifier in models["categories"]:
    matching_model =
    models["categories"][performance_classifier][0]
    with open(os.path.abspath(matching_model), 'rb')
    as file:
        trained_model = pickle.load(file)
    self.performance_classifier =
    PerformanceClassifier(trained_model)

self.syscallCollector = []
self.performanceCollector = []

def handle_message(self, msg, classifier_type):
    if classifier_type == 'syscalls':
        self.process_data(msg, "syscalls")
    elif classifier_type == 'performance':
        self.process_data(msg, "performance")
    else:
        print("Classifier_Error")

#collect data into timeslices, transform and make prediction
def process_data(self, data, classifier_type):
    if classifier_type == "syscalls":
        #syscall data preprocessing
    elif classifier_type == "performance":
        #resource usage data preprocessing
    else:
        print("Data_Processing_Error")

```

The classifier manager is initialized via a websocket request when such is received from the front-end. The models which were selected in the front-end are loaded for the system call and resource usage classifiers in the backend, and the classifier manager class becomes ready to receive system call and resource usage data from the system monitors. When such data is received, it is preprocessed according to the methods outlined in the design section

(for data format samples, see the "Implementing the Model Trainer" section above), and the processed data is forwarded to the backend models, which return a prediction.

5.3.2 Anomaly Detector Manager

The anomaly detector manager follows a similar structure as above, although lacking a feature extractor and having different websocket namespaces. Therefore, it has been omitted.

5.4 Extending the Front-End Interface

The front-end interface was changed in several ways. In particular:

1. A model selector section was added to the sandbox start interface.
2. Two malware analysis tables were added to the real-time analysis dashboard

5.4.1 Model Selector

A model selector was added to the "Start New Sandbox" dialog, allowing to specify pre-trained models for each of the data (system calls/resource usage) and model type (classifier/anomaly detector) combinations.

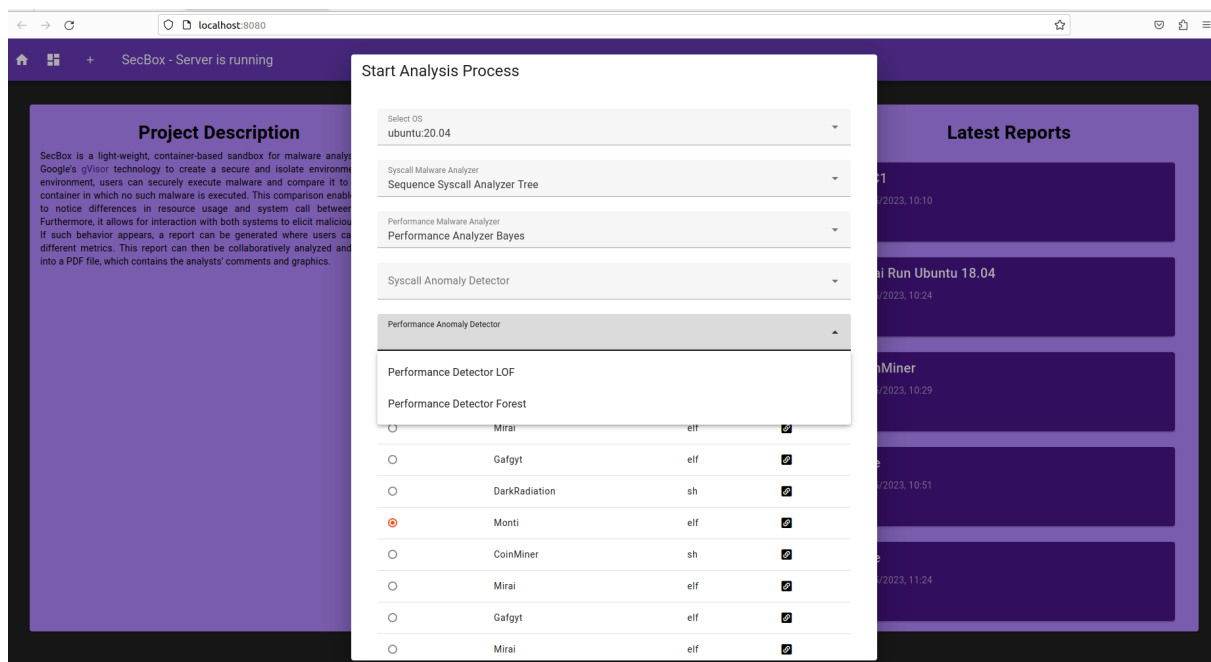


Figure 5.6: The model selector dropdowns in the start menu

Each of these models would then generate predictions regarding whether the sandbox was infected or not in real-time, once the sandbox was launched.

5.4.2 Malware Analysis Tables

2 malware analysis tables were added to the real-time sandbox interface, see Figure 5.7.

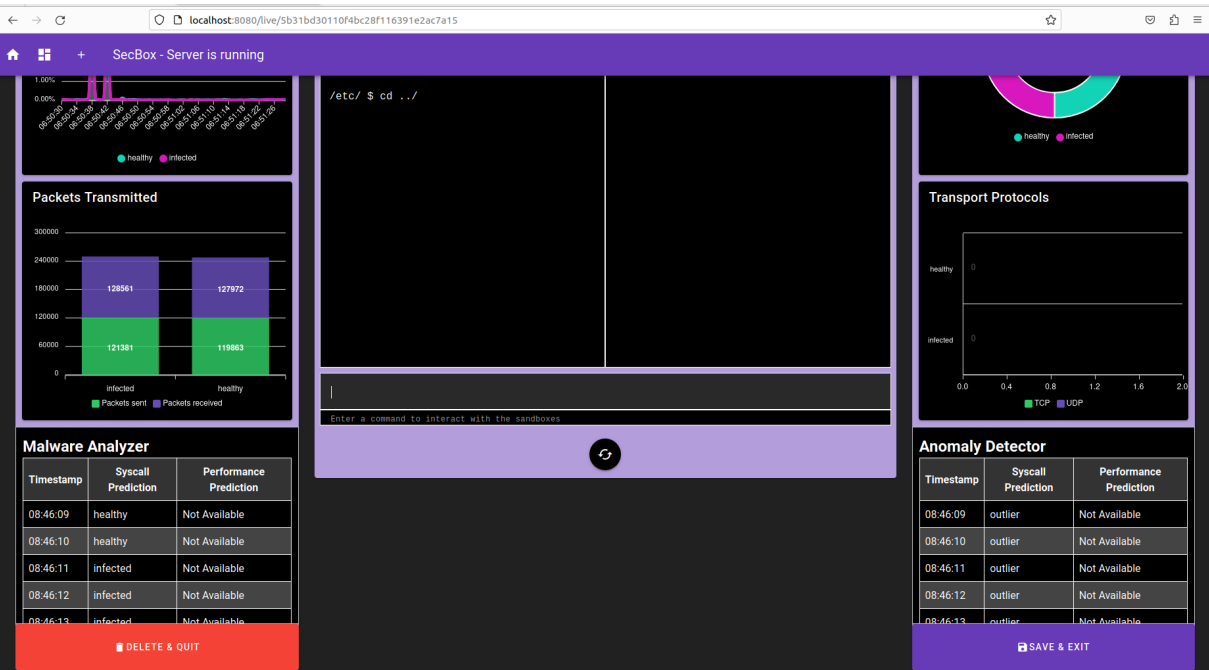


Figure 5.7: Malware analysis tables as part of the front-end interface

On the bottom-left of the dashboard is the Malware Analyzer table, which receives predictions from the system call classifier and resource usage classifier. The predictions are of the format

(timestamp, prediction)

and are matched together in the front-end in order to have predictions from both models for the same timestamp easily visible.

On the bottom-right of the dashboard is the Anomaly Detector table, which receives predictions from the system call and resource usage anomaly detectors, and matches them together as described above. The displayed predictions are then either "inlier" if a data point was perceived as normal, or "outlier" if it was perceived to be malicious.

Chapter 6

Prototype Evaluation

This chapter conducts an evaluation of the implemented prototype solution, in order to assess the resource efficiency and effectiveness of the different machine learning models

6.1 Generating the Test Data Set

To generate the test data set, a machine with an Intel i7-8750H CPU (6 cores, 2 threads per core) and 16GB of RAM was used, running Ubuntu 22.04.

A zip archive was downloaded from [31] into the SecBox environment (zipped size 176MB, unzipped size 412MB). This archive contained various data formats, such as text files, csv files, powerpoint presentations and pdf files (see Figure 6.1. It was cloned 12 times, as downloading larger data sets proved to be infeasible due to SecBox's bandwidth limitations, with the cloning resulting in a total size of 4944MB for the files.

```

infected

/etc/ $ cd 026

/etc/026/ $ ls
026000.ppt 026001.ppt 026002.ppt 026003.ppt
026004.xls 026005.ppt 026006.ppt 026007.ppt
026008.ppt 026009.ppt 026010.ppt 026011.ppt
026013.ppt 026014.ppt 026015.ppt
026016.html 026017.ppt 026018.ppt
026019.ppt 026020.ppt 026021.ppt 026022.xls
026023.ppt 026024.xls 026025.ppt 026026.ppt
026027.xls 026028.ppt 026029.ppt 026030.ppt
026031.ppt 026032.pdf 026033.pdf 026034.jpg
026035.gif 026036.jpg 026037.pdf 026038.pdf
026039.csv 026040.pdf 026041.pdf 026042.pdf
026043.pdf 026044.pdf 026045.pdf 026046.pdf
026047.csv 026048.pdf 026049.csv 026050.pdf
026051.csv 026052.pdf 026053.xml 026054.xml
026055.pdf 026056.html 026057.pdf
026058.pdf 026059.log 026060.log 026061.pdf
026062.xls 026063.xls 026064.xls 026065.gz
026066.pdf 026067.pdf 026068.xls 026069.xls
026070.pdf 026071.gif 026072.xls 026073.xls
026074.xls 026075.xls 026076.csv 026077.xls
026078.xls 026079.xls 026080.dbase3
026081.xls 026082.xls 026083.xls 026084.xls
-----

```

Figure 6.1: The file contents of the downloaded archive

These files were placed under the `/etc/ SecBox` directory (Figure 6.1), and a Monti ransomware file was then run on an Ubuntu 22.04 operating system within the SecBox environment, with the exact start time of malware execution being recorded.

```
infected
egg33e1195e023e007de1.e11 etc home lib
lib32 lib64 libx32 malware.7z media mnt opt
proc root run sbin srv sys tmp usr var

/ $ cd etc

/etc/ $ ls
026 027 028 029 030 031 032 033 034 035 036
037 038 adduser.conf alternatives apt
bash.bashrc bindresvport.blacklist ca-
certificates ca-certificates.conf cloud
cron.d cron.daily debconf.conf
debian_version default deluser.conf dpkg
e2scrub.conf environment fstab gai.conf
group gshadow gss host.conf hostname hosts
init.d issue issue.net kernel ld.so.cache
ld.so.conf ld.so.conf.d legal libaudit.conf
login.defs logrotate.d lsb-release machine-
id mke2fs.conf mtab netconfig networks
nsswitch.conf opt os-release pam.conf pam.d
passwd profile profile.d rc0.d rc1.d rc2.d
rc3.d rc4.d rc5.d rc6.d rcS.d resolv.conf
rmt security selinux shadow shells skel ssl
subgid subuid sysctl.conf sysctl.d systemd
terminfo update-motd.d wgetrc xattr.conf
```

Figure 6.2: /etc/ folder contents after archive cloning

Once all the test files were encrypted, the sandbox execution was stopped (15 minutes of execution time), and the system call (174MB) and resource usage (34MB) files were downloaded onto the machine (Figure 6.3). Note that the resource usage files were not available from the SecBox front-end download interface, so they were taken directly from the connected database storage.

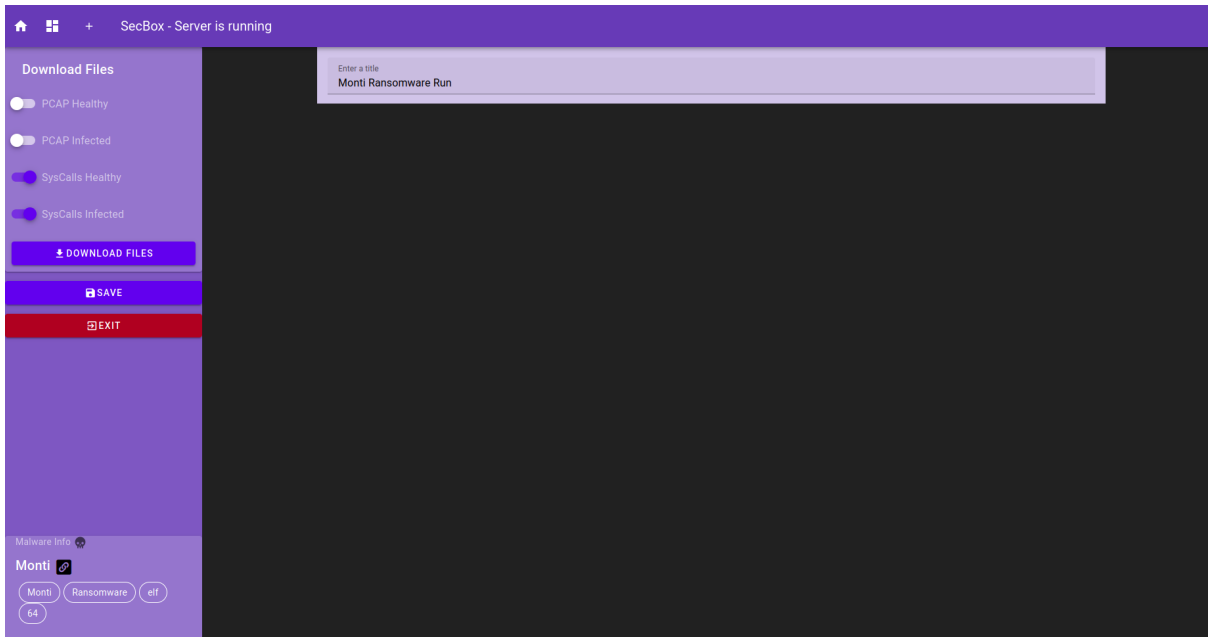


Figure 6.3: Downloading the files resulting from a Monti execution using the SecBox interface

The file versions from the "infected" sandbox were cut to start from the malware execution timestamp, rather than the sandbox setup time, in order to avoid labeling the healthy sandbox state prior to malware execution as "infected". This method allowed to separate the "infected" section of the data into its own file, with the data from both the system call and resource usage files then being passed to the model trainers, which were executed while being provided the data files to train and store their respective machine learning models.

Once preprocessed according to the methods outlined in the Design chapter, the system call file had 82 features (total of system calls considered), while the resource usage file had 4 features (cpu percentage, ram, packages sent, packages received)

6.2 Model Resource Efficiency

The resource efficiency of each model can be measured by looking at the training time and resident set size (a metric that measures how much RAM a process is actively using). The resource efficiency table can be seen in Table 6.1.

Table 6.1: Model Resource Efficiency Table

Model	Training Time (s)	Training RSS (KB)
Resource Usage Classifier Bayes	0.07	108540
Resource Usage Classifier Tree	0.07	108420
Resource Usage Detector LOF	0.07	111168
Resource Usage Detector Forest	0.17	111240
System Call Classifier Frequency Bayes	2.51	342372
System Call Classifier Sequence Bayes	244.40	342732
System Call Classifier Frequency Tree	2.52	342220
System Call Classifier Sequence Tree	253.70	342816
System Call Detector Frequency LOF	2.27	329420
System Call Detector Sequence LOF	167.82	297024
System Call Detector Frequency Forest	2.18	299572
System Call Detector Sequence Forest	168.90	298284

It can be seen that the resource usage models were much faster to train compared to the system call models, and required much fewer active RAM.

This is due to their simplistic data processing (combining different metrics like CPU percentage and RAM usage into a single array for each timestamp) and smaller file sizes (in general, a JSON resource usage data file was at least 5 times smaller compared to a system call CSV file).

On the other hand, system call models needed more resources, with an interesting observation - models using sequence-based preprocessing required roughly 100 times more training time than models using frequency-preprocessed data (despite RSS requirements remaining the same). This is because of the data structure arising from sequence preprocessing (see Table 5.2) - a list of lists, with $O(n^2)$ space complexity, which is much worse than the $O(n)$ complexity of the frequency preprocessed data structure (see 5.1).

6.3 Model Effectiveness

To evaluate the usefulness of the different models when outputting predictions, a model from each category was selected:

1. System Call Classifier (Frequency Preprocessed) Naive Bayes
2. Resource Usage Classifier Decision Tree
3. System Call Anomaly Detector (Sequence Preprocessed) LOF
4. Resource Usage Anomaly Detector Isolation Forest

These models were used to output predictions in a test run against a Monti ransomware sample, and a confusion matrix was created for each model. Note that the sandbox setup phase was ignored for these results:

Table 6.2: Confusion Matrix - System Call Classifier with Naive Bayes

		Predicted Outcome		total
		p	n	
Actual Outcome	p	29	0	29
	n	3	33	36
total		32	33	

Table 6.3: Confusion Matrix - Resource Usage Classifier with Decision Tree

		Predicted Outcome		total
		p	n	
Actual Outcome	p	0	16	16
	n	0	3	3
total		0	19	

Table 6.4: Confusion Matrix - System Call Anomaly Detector with Local Outlier Factor

		Predicted Outcome		total
		p	n	
Actual Outcome	p	29	0	29
	n	36	0	36
total		65	0	

Table 6.5: Confusion Matrix - Resource Usage Anomaly Detector with Isolation Forest

		Predicted Outcome		
		p	n	total
Actual Outcome	p	16	0	16
	n	3	0	3
total		19	0	

Based on these matrices, further metrics can be calculated:

6.3.1 Accuracy

Accuracy is the ratio of correct predictions against the total number of predictions.

Table 6.6: Model Accuracy Table

Model	Model Accuracy
System Call Classifier (Frequency Preprocessed) Naive Bayes	0.9538
Resource Usage Classifier Decision Tree	0.1579
System Call Anomaly Detector (Sequence Preprocessed) LOF	0.4461
Resource Usage Anomaly Detector Isolation Forest	0.8421

6.3.2 Precision

Precision measures how many of the predicted "positive" instances are positive in reality.

Table 6.7: Model Precision Table

Model	Model Precision
System Call Classifier (Frequency Preprocessed) Naive Bayes	0.9063
Resource Usage Classifier Decision Tree	0.0
System Call Anomaly Detector (Sequence Preprocessed) LOF	0.4461
Resource Usage Anomaly Detector Isolation Forest	0.8421

6.3.3 Recall

Recall is a metric that measures how many of the actual positive data points were predicted correctly by the model.

Table 6.8: Model Recall Table

Model	Model Recall
System Call Classifier (Frequency Preprocessed) Naive Bayes	1.0
Resource Usage Classifier Decision Tree	0.0
System Call Anomaly Detector (Sequence Preprocessed) LOF	1.0
Resource Usage Anomaly Detector Isolation Forest	1.0

6.3.4 F1-Score

F1-Score is a metric combining precision and recall, to give a more balanced perspective of the model by considering both false positives and false negatives in its evaluation.

Table 6.9: Model F1-Score Table

Model	Model F1-Score
System Call Classifier (Frequency Preprocessed) Naive Bayes	0.9508
Resource Usage Classifier Decision Tree	Undefined
System Call Anomaly Detector (Sequence Preprocessed) LOF	0.6170
Resource Usage Anomaly Detector Isolation Forest	0.9142

6.3.5 Interpreting the Numerical Results

Overall, it can be seen that the first model (system call classifier with naive bayes) showed great performance across all metrics when generating predictions for a possible Monti infection.

Unfortunately, the second model (resource usage classifier with decision tree) did not perform nearly as well, and was unable to detect the Monti infection.

This is likely due to the SecBox architecture and how it relates to the raw data required by the models - for system calls, this raw data is taken directly from the system call monitor running on the host machine, whereas the corresponding resource usage data is not available from the performance monitor - instead, the only place it can currently be taken is from a function that prepares it for front-end graphing, which may have led to longer delays for data processing and unexpected data groupings.

Alternatively, it could also be due to the breadth of metrics considered - during Monti execution, the only noticeable spike was in CPU usage, whereas the other system resources such as RAM remained at reasonable levels. As a result, the change in CPU usage may not have been noticeable enough for the model.

Finally, the last two models (system call anomaly detector with LOF and resource usage anomaly detector with isolation forest) exhibited unusual behavior, labelling nearly every data sample as an outlier.

There is a straightforward explanation for this - the data sets using which the models were trained are relatively limited, meaning that the possible "normal" behaviors of the system are not described sufficiently well to the machine learning models.

This situation causes the anomaly detector to interpret even slight deviations from the norm as unusual, so when the new malware sample was executed, every data point was labelled as an outlier.

In order to solve this, the representation of "normal" behavior within the SecBox will need to be extended, either through larger data sets in order to have a sufficient number of data points for patterns to be discovered, or through more varied representations of the healthy sandbox behaviors, for example by executing code that requires a lot of system resources, so as to normalize such behavior.

6.4 Model Reproducibility and Explainability

The final evaluation criteria is the reproducibility and explainability of the trained machine learning models.

When looking at model reproducibility, the set objective has been fulfilled, because every created model is linked to a configuration file, which details every input that the model received from the user, as well as additional information such as the random seed chosen by the previous model (see Figure 6.4). Using this file, any model can be reconstructed from scratch to generate the same predictions, therefore making the training process reproducible.

```
backend > modelTrainer > models > {} monti_performance_detector_lof.json > ...
1  {
2    "modelName": "monti_performance_detector_lof",
3    "modelType": "LocalOutlierFactor",
4    "dataFilePaths": [
5      "backend/modelTrainer/trainingData/montiPerformance.json"
6    ],
7    "randomState": "N/A",
8    "contamination": 0.1,
9    "novelty": true,
10   "n_neighbors": 20,
11   "timePeriod": 1000000000
12 }
```

Figure 6.4: A JSON model configuration file

In terms of explainability, the objective has been partially fulfilled, because the chosen machine learning models have a high degree of explainability (for a discussion of this, see 4.1.2), however, these metrics are not currently visually displayed anywhere, meaning that a user that wishes to access them would have to look for these metrics themselves within the model trainer program module.

Chapter 7

Summary and Conclusions

This chapter summarises the work that was conducted during this project, concludes the main findings of this project, and proposes ideas for how the project can be extended in the future.

7.1 Summary of Work Conducted and Conclusion of Main Findings

The summary of work is done on a per-chapter basis, including the main findings within each chapter:

7.1.1 Introduction

Within the introduction section, the current situation in the world with regards to technology usage and cybersecurity was looked into, and it was found that small and midsize enterprises (SMEs) were becoming increasingly reliant on technology, yet lacked appropriate cybersecurity measures, leaving them prone to cyberattacks from malicious actors.

The usage of machine learning for malware analysis was established as a promising technique to resolve this problem, and the need for a machine learning malware analysis tool was explained.

7.1.2 Background

Malware analysis was looked into in detail, and it was found that several techniques exist. The best option was found to be a combination of static analysis conducted on the artifacts produced by the malware, and dynamic analysis of the malware during its runtime. This was because it offered the best chances of deciphering the malware's purpose and functions.

It was also found that for any machine learning model, a machine learning pipeline needs to be configured, establishing the need to make choices regarding data preprocessing, model selection and model evaluation methods.

7.1.3 Related Work

In related work, two distinct categories of works were looked into - machine learning cybersecurity solutions, and sandbox systems with machine learning components.

For machine learning cybersecurity solutions, it was determined that existing solutions lacked important aspects for the project, such as reproducibility of outcomes, accountability with regards to why certain predictions were made, and the option to implement different machine learning pipelines.

For sandbox systems, it was found that most sandboxes that implemented machine learning components were closed source, thus were not suitable for the project. However, SecBox was discovered as an appropriate system that fulfilled the basic requirements - being actively maintained, accountable and open source, although lacking a machine learning component.

As a result, it was decided to extend SecBox with a machine learning component in order to fulfill the project's goals.

7.1.4 Design

Within the design section, the system requirements for the prototype solution were outlined, and it was decided to use the available system call CSV files and resource usage JSON files (which are generated with every run), in order to provide suitable data to the machine learning component.

Choices regarding machine learning algorithms were made, with Decision Tree and Naive Bayes models chosen for malware classification, while Isolation Forest and Local Outlier Factor were chosen for anomaly detection. These choices were made due to the fact that all of these models can be trained in a reproducible manner, and possess a good degree of explainability in terms of the predictions that they output.

Data preprocessing methods were decided upon, with the proposal to use frequency-based and sequence-based data preprocessing for the system call files, while simple data aggregation was used for the resource usage files.

Lastly, the general architecture of the project was outlined, extending the existing SecBox architecture with a new model trainer module in the backend, new data managers were added, with the capability to command the model trainers via websocket connections, and new front-end components were designed, intended to display the model predictions in real-time.

7.1.5 Implementation

For the implementation, the initial setup of the SecBox environment is described, followed by the results of initial test runs of various malware types in the SecBox environment. It was found that certain malware types (such as Monti ransomware) executed as expected within the environment, while others (such as available versions of coin miners) did not execute at all due to missing libraries.

A decision was made to focus on the Monti ransomware family due to instability of other malware samples, and the more apparent effect of the ransomware in terms of aggressive resource usage providing better training data for the models.

Lastly, the concrete implementation of the Model Trainer module, and extensions to the Data Manager module are described, as well as the additions to the front-end interface such as the prediction tables and model chooser interface.

7.1.6 Evaluation

Finally, in the evaluation section, confusion matrices were created for selected machine learning models, and then used to calculate metrics such as accuracy, precision, recall and F1-Score for each model in order to assess how effective it is at recognizing Monti ransomware.

The resource usage of each model was measured and explained, both for training the model using the available data files, as well as generating predictions regarding infected files in real-time.

Lastly, aspects such as the reproducibility and explainability of each model were looked into, and the overall findings for each model were explained in terms of the SecBox architecture and the chosen machine learning pipelines.

7.2 Further Work

There are several aspects in this project that were touched upon, but are significant enough to warrant additional research in the future.

7.2.1 Realistic Representation of Sandbox Behavior

One of the problems that was encountered during this project (and explained in the Evaluation section) was the lack of realistic representation of sandbox behaviors under load.

The Monti ransomware uses the majority of a system's resources during its execution, particularly with regards to the CPU percentage usage. On the other hand, the healthy

sandbox environment does not execute any tasks by default, meaning that, if an action is performed in the future that causes high load on the CPU, it will be classified as a Monti virus by the current models.

An attempt to rectify this was made by simulating load through downloading large zip files, unzipping them and moving their contents within the SecBox environment, however this proved insufficient, possibly due to the differences in the character of system calls and level of CPU usage.

Therefore, it will be of great benefit if a technique to model "healthy" system load is designed, as it would help decrease the number of false positive predictions by the machine learning models.

7.2.2 Changing how the Resource Usage Data is Provided to the Model Trainer

It was also mentioned in the evaluation that an issue arose with the resource usage data during the real-time prediction phase of the SecBox - some of the data was very delayed in arriving at the model trainer class (when compared to the system call data), leading to worse predictions by the machine learning models.

In order to resolve this issue, the existing SecBox resource usage monitor implementation could be changed to be consistent with the system call monitor, because currently, the necessary system call data can be taken directly from the system call monitor, however, the corresponding resource usage data is only available from a data processing function within the websocket implementation of the project.

To resolve this issue, a data processing function that provides all the needed parameters can be added directly to the resource usage monitor, minimizing delays and data format changes due to data processing for the front-end, and therefore improving the quality of the models through better data quality and availability.

7.2.3 Training the Models from the Front-End

Currently, the machine learning models are trained and stored by running a training script contained within the Model Trainer python files in the backend. From the user point of view, it would make sense to integrate the model training into the front-end, allowing users to select data files and train models directly from the SecBox interface.

This can be achieved by extending the websocket connections (within the WebAPI python file) with an additional namespace, which would allow the front-end to issue commands such as "train a decision tree classifier using this data file" to the model trainer in the backend, and would make it easier for users to experiment with training new models.

7.2.4 Integrating the Models' Explainable Metrics into the Front-End Interface

Lastly, the evaluation mentioned that while the chosen machine learning models contain a good degree of explainability, the benefits are not being maximized at the moment, as the provided metrics are not shown to the users. For example, the decision tree and naive bayes models contain feature importance ranking data, showing which features are considered to be more relevant to the problem at hand. Likewise, the Local Outlier Factor algorithm calculates an anomaly score for each data point, and the Isolation Forest algorithm calculates an isolation score.

Therefore, a method of displaying these underlying scores could be designed for each model, providing users with this information in the front-end during the sandbox execution, and therefore allowing them to better understand why a certain prediction was made by the model with regards to a file.

Bibliography

- [1] L. Vermeer and M. Thomas, “Pharmaceutical/high-tech alliances; transforming health-care? digitalization in the healthcare industry”, *Strategic Direction*, vol. 36, no. 12, pp. 43–46, 2020.
- [2] Y. Sun, W. Wang, Y. Chen, and Y. Jia, “Learn how to assist humans through human teaching and robot learning in human-robot collaborative assembly”, *IEEE Systems Journal*, vol. 52, no. 2, pp. 728–738, 2022.
- [3] M. S. Ismail, M. I. Mohd Salmi Md Noorani, F. A. Razak, and M. A. Alias, “Predicting next day direction of stock price movement using machine learning methods with persistent homology: Evidence from kuala lumpur stock exchange”, *Applied soft computing*, vol. 93, p. 106422, 2020.
- [4] M. Cohen and J. Smetzer, “Understanding human over-reliance on technology; it’s exelan, not exelon; crash cart drug mix-up; risk with entering a “test order””, *Hospital pharmacy (Philadelphia)*, vol. 52, no. 1, pp. 7–12, 2017.
- [5] A. Alahmari and B. Duncan, “Cybersecurity risk management in small and medium-sized enterprises: A systematic review of recent evidence”, in *2020 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, 2020, pp. 1–5. DOI: 10.1109/CyberSA49311.2020.9139638.
- [6] A. Zimba and M. Chishimba, “On the economic impact of crypto-ransomware attacks: The state of the art on enterprise systems”, *European journal for security research*, vol. 4, no. 1, pp. 3–31, 2019.
- [7] E. Gandotra, D. Bansal, and S. Sofat, “Zero-day malware detection”, in *2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*, 2016, pp. 171–175. DOI: 10.1109/ISED.2016.7977076.
- [8] F. Abri, S. Siامي-Namini, M. A. Khanghah, F. M. Soltani, and A. S. Namin, “Can machine/deep learning classifiers detect zero-day malware with high accuracy?”, in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 3252–3259. DOI: 10.1109/BigData47090.2019.9006514.
- [9] M. Vasilescu, L. Gheorghe, and N. Tapus, “Practical malware analysis based on sandboxing”, in *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*, 2014, pp. 1–6. DOI: 10.1109/RoEduNet-RENAM.2014.6955304.
- [10] M. Maass, A. Sales, B. Chung, and J. Sunshine, “A systematic analysis of the science of sandboxing”, *PeerJ Computer Science*, vol. 2, e43, 2016.

- [11] L. Vokorokos, A. Baláž, and B. Madoš, “Application security through sandbox virtualization”, *Acta Polytechnica Hungarica*, vol. 12, no. 1, pp. 83–101, 2015.
- [12] J. v. d. Assen, A. H. Celdrán, A. Zermin, R. Mogenicato, G. Bovet, and B. Stiller, “Secbox: A lightweight container-based sandbox for dynamic malware analysis”, in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–3. DOI: 10.1109/NOMS56928.2023.10154293.
- [13] H. A. Noman, Q. Al-Maatouk, and S. A. Noman, “A static analysis tool for malware detection”, in *2021 International Conference on Data Analytics for Business and Industry (ICDABI)*, 2021, pp. 661–665. DOI: 10.1109/ICDABI53623.2021.9655866.
- [14] A. V, V. P, V. G. Menon, *et al.*, “Malware detection using dynamic analysis”, in *2023 International Conference on Advances in Intelligent Computing and Applications (AICAPS)*, 2023, pp. 1–6. DOI: 10.1109/AICAPS57044.2023.10074588.
- [15] R. B. Hadiprakoso, H. Kabetta, and I. K. S. Buana, “Hybrid-based malware analysis for effective and efficiency android malware detection”, in *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, 2020, pp. 8–12. DOI: 10.1109/ICIMCIS51567.2020.9354315.
- [16] N. Balram, G. Hsieh, and C. McFall, “Static malware analysis using machine learning algorithms on apt1 dataset with string and pe header features”, in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 90–95. DOI: 10.1109/CSCI49370.2019.00022.
- [17] S. Gulmez, A. G. Kakisim, and I. Sogukpinar, “Analysis of the dynamic features on ransomware detection using deep learning-based methods”, in *2023 11th International Symposium on Digital Forensics and Security (ISDFS)*, 2023, pp. 1–6. DOI: 10.1109/ISDFS58141.2023.10131862.
- [18] A. Posoldova, “Machine learning pipelines: From research to production”, *IEEE Potentials*, vol. 39, no. 6, pp. 38–42, 2020. DOI: 10.1109/MPOT.2020.3016280.
- [19] S. Sedkaoui, “Supervised versus unsupervised algorithms: A guided tour”, in *Data Analytics and Big Data*. 2018, pp. 123–151. DOI: 10.1002/9781119528043.ch7.
- [20] I. Bilbao and J. Bilbao, “Overfitting problem and the over-training in the era of data: Particularly for artificial neural networks”, in *2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*, 2017, pp. 173–177. DOI: 10.1109/INTELCIS.2017.8260032.
- [21] V. Pintiysky, “System and method of analysis of files for maliciousness in a virtual machine”, US10339301, Jul. 2019.
- [22] C. Compton, “Cuckoo sandbox: Automated malware analysis.”, Jul. 2019.
- [23] T. Lengyel, *Drakwuf wiki*, 2022.
- [24] K. Monappa, *Limon - sandbox for analyzing linux malwares*, 2015.
- [25] ANY.RUN, *Any.run dynamic malware analysis sandbox*, Available at <https://app.any.run/>, 2023.
- [26] W. Wang, F. Harrou, Y. Sun, B. Bouyeddou, and S.-M. Senouci, “A stacked deep learning approach to cyber-attacks detection in industrial systems: Application to power system and gas pipeline systems”, *Cluster Comput*, vol. 25, pp. 561–578, 2022. DOI: <https://doi.org/10.1007/s10586-021-03426-w>.

- [27] K. Rieck, P. Trinius, C. Willems, and T. Holz, “Automatic analysis of malware behavior using machine learning”, *Journal of computer security*, vol. 19, no. 4, pp. 639–668, 2011.
- [28] S. Varma and J. Narasimharao, “Malware analysis with machine learning: Classifying malware based on pe header”, *International Journal for Research in Applied Science and Engineering Technology (IJRASET)*, vol. 10, no. VI, pp. 3583–3590, 2022. DOI: <https://doi.org/10.22214/ijraset.2022.44668>.
- [29] M. S. Akhtar and T. Feng, “Malware analysis and detection using machine learning algorithms”, *Symmetry*, vol. 14, no. 11, p. 2304, 2022. DOI: <https://doi.org/10.3390/sym14112304>.
- [30] L. C. Navarro, A. K. W. Navarro, A. Gregio, A. Rocha, and R. Dahab, “Leveraging ontologies and machine-learning techniques for malware analysis into android permissions ecosystems”, *Computers & Security*, vol. 78, pp. 429–453, 2018. DOI: <https://doi-org/10.1016/j.cose.2018.07.013>.
- [31] *Dataset 26*, <https://digitalcorpora.s3.amazonaws.com/corpora/files/govdocs1/zipfiles/026.zip>.

Abbreviations

CSV	Comma-Separated Values
CPU	Central Processing Unit
DT	Decision Tree
JSON	JavaScript Object Notation
LOF	Local Outlier Factor
ML	Machine Learning
RAM	Random Access Memory
RSS	Resident Set Size
SMEs	Small to Midsize Enterprises
Syscall	System Call

Glossary

Machine Learning A process of using historical data to create a prediction algorithm for future data

Malware Analysis The study of the functionality, purpose, origin, and potential impact of malicious software

Sandboxing A technique used in cybersecurity, where an application can be encapsulated within the underlying host machine

List of Figures

4.1	SecBox existing architecture	20
4.2	SecBox post-mortem analysis tab	21
4.3	SecBox existing analysis page	22
5.1	A configured .env file for the SecBox api	25
5.2	SecBox error when attempting to run a CoinMiner virus - various libraries not found	26
5.3	Monti ransomware in action within SecBox - note the CPU usage spike and encrypted disk files	27
5.4	Example system call CSV file	28
5.5	Sample stored models as part of the modelTrainer module	32
5.6	The model selector dropdowns in the start menu	34
5.7	Malware analysis tables as part of the front-end interface	35
6.1	The file contents of the downloaded archive	38
6.2	/etc/ folder contents after archive cloning	39
6.3	Downloading the files resulting from a Monti execution using the SecBox interface	40
6.4	A JSON model configuration file	45

List of Tables

3.1	Comparison of Sandbox Systems	13
3.2	Comparison of Machine Learning Cybersecurity Solutions	15
5.1	Sample frequency-processed system call list	28
5.2	Sample sequence-processed system call list	29
5.3	Resource usage processed data	29
6.1	Model Resource Efficiency Table	41
6.2	Confusion Matrix - System Call Classifier with Naive Bayes	42
6.3	Confusion Matrix - Resource Usage Classifier with Decision Tree	42
6.4	Confusion Matrix - System Call Anomaly Detector with Local Outlier Factor	42
6.5	Confusion Matrix - Resource Usage Anomaly Detector with Isolation Forest	43
6.6	Model Accuracy Table	43
6.7	Model Precision Table	43
6.8	Model Recall Table	44
6.9	Model F1-Score Table	44

Appendix A

Installation Guidelines

Follow the installation guidelines of the underlying SecBox system (provided as a README file within the code). The database setup step (marked as being optional for the original system) is recommended, so as to allow retrieval of the resource usage files for model training.