applied sciences

MDPI

Article

Machine Learning and Deep Learning Based Model for the Detection of Rootkits Using Memory Analysis

Basirah Noor *,† and Sana Qadir *,†

School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad 44000, Pakistan

* Correspondence: bnoor.msis19seecs@seecs.edu.pk (B.N.); sana.qadir@seecs.edu.pk (S.Q.)

† These authors contributed equally to this work.

Abstract: Rootkits are malicious programs designed to conceal their activities on compromised systems, making them challenging to detect using conventional methods. As the threat landscape continually evolves, rootkits pose a serious threat by stealthily concealing malicious activities, making their early detection crucial to prevent data breaches and system compromise. A promising strategy for monitoring system activities involves analyzing volatile memory. This study proposes a rootkit detection model that combines memory analysis with Machine Learning (ML) and Deep Learning (DL) techniques. The model aims to identify suspicious patterns and behaviors associated with rootkits by analyzing the contents of a system's volatile memory. To train the model, a diverse dataset of known rootkit samples is employed, and ML and deep learning algorithms are utilized. Through extensive experimentation and evaluation using SVM, RF, DT, k-NN, and LSTM algorithms, it is determined that SVM achieves the highest accuracy rate of 96.2%, whereas Execution Time (ET) shows that k-NN depicts the best performance, and LSTM (a DL model) shows the worst performance among the tested algorithms. This research contributes to the development of advanced defense mechanisms and enhances system security against the constantly evolving threat of rootkit attacks.

Keywords: memory analysis; rootkits; deep learning; machine learning; execution time

Citation: Noor, B.; Qadir, S. Machine Learning and Deep Learning Based Model for the Detection of Rootkits Using Memory Analysis. *Appl. Sci.* **2023**, *13*, 10730. https://doi.org/10.3390/app131910730

Academic Editors: Howon Kim and Thi-Thu-Huong Le

Received: 20 July 2023
Revised: 29 August 2023
Accepted: 4 September 2023
Published: 27 September 2023

1. Introduction

A rootkit is a malicious program with a highly deceptive nature. It operates by concealing its presence in the system and enabling unauthorized root access to attackers, allowing then to gain complete control over the compromised system. Rootkits are designed to evade detection and this makes it challenging for anti-malware tools to detect their infiltration. Once installed, a system may exhibit unusual behaviour, indicating remote access by an attacker. Rootkits are particularly hazardous and can lead to significant data loss and damage.

The term "rootkit" is derived from the combination of two words, "root" and "kit". In UNIX and Linux environments, "root" refers to the system administrator who possesses the highest level of access. "Kit" denotes a collection of tools and techniques. Consequently, a rootkit is defined as a set of tools or techniques enabling unauthorized individuals to gain and sustain administrator-level access during an attack while evading detection by authorized users and administrators [1].

Rootkits have been a significant concern for security engineers since the late 1980s. Initially, they were primarily used to hide log files and application binaries. The early generation of these malicious software targeted user-level programs, making them relatively easy to detect using simple checksum methods. However, the threat landscape has evolved, and modern rootkits pose more and more danger, as operating systems are now widely employed in various devices such as smartphones, IoT nodes, computers,

and embedded devices [2,3]. Consequently, researchers are actively working on developing efficient methods to detect these sophisticated rootkits that can evade traditional detection techniques.

The frequency of rootkit attacks is also on the rise. According to reports from Avast, a leading cybersecurity company, the number of users affected by rootkits increased from 10,000 in 2020 to 100,000 in 2021. Therefore, the main objective of this research is to devise a reliable and efficient detection technique specifically tailored for rootkits that are challenging to detect.

Rootkits are designed to conceal themselves from system administrators and users, allowing them to operate covertly and pose significant security threats. They pose a threat not only to the host systems, but also to the virtual machines [4]. Therefore, there is a crucial requirement for a standardized mechanism that can efficiently detect rootkits. This research aims to investigate the effectiveness of memory analysis techniques for rootkit detection. Additionally, we aim to develop an automated tool utilizing Machine Learning (ML) and Deep Learning (DL) for the quick and accurate identification of rootkits within a system.

To overcome the limitations of previous research, this paper sets out to achieve the following objectives:

- Introduce an effective technique that leverages memory analysis to detect concealed rootkits;
- Develop models and tools to automate the analysis of memory dumps for efficient rootkit detection;
- Investigate and explore a novel set of features extracted from memory images, including DLLs, handles, privileges, network connections, modules, injections, and services.

By fulfilling these objectives, this research aims to contribute to the advancement of rootkit detection methods and overcome the existing limitations in the field.

The rest of the paper is organized as follows: Section 2 provides a background on basics of ML and DL. Section 3 presents the related studies on malware and rootkit detection. The proposed rootkit detection approach and the memory-based dataset are described in Section 4. Section 5 explains the experimental results and discuss them and compares them with the existing literature. Finally, the conclusion and future works are presented in Section 6 .

2. Background

Memory analysis is a critical aspect of cybersecurity, involving the examination of a computer's volatile memory to identify malicious activities, unauthorized processes, and potential threats. It has emerged as a promising approach for detecting and classifying malware, surpassing the limitations of static and behavioral analysis. Traditionally, memory analysis has been a time-consuming and intricate manual process, requiring expert analysts to sift through memory dumps and detect anomalies. However, the integration of ML and DL techniques has revolutionized this field, enabling the automation of memory analysis for faster and more accurate threat detection. More precisely, the benefits of this automation include:

- ML- and DL-powered automation significantly accelerate memory analysis, allowing security teams to detect threats faster and allocate resources more effectively.
- These technologies reduce the risk of human error and increase the accuracy of threat detection by considering a wider range of patterns and behaviors.
- Automated memory analysis can handle a high volume of memory dumps simultaneously, making it suitable for large-scale environments.
- ML and DL models can provide real-time or near-real-time analysis of memory dumps, enabling a rapid response to ongoing attacks.

2.1. Machine Learning (ML) in Memory Analysis

ML algorithms can analyze memory dumps and recognize patterns associated with malware, rootkits, and other malicious activities. it has been used for over a past decade

to detect the malwares [5]. By training on labeled data, ML models can learn to identify anomalies and deviations from normal system behavior [6]. These algorithms can detect memory-resident malware, unauthorized process injection, and other stealthy attacks by detecting deviations from learned behavioral profiles [7]. For instance, ML algorithms such as Random Forest (RF) or Support Vector Machine (SVM) can be trained on features extracted from memory dumps, such as API calls, memory sections, and process relationships. These models can then classify memory contents as malicious or benign, aiding in the rapid identification of threats. A short explanation of four common ML algorithms is given below:

- **Random Forest (RF):** Random Forest is an ensemble algorithm that combines multiple decision trees to enhance accuracy and prevent over-fitting [8]. It creates diverse decision trees and aggregates their predictions to improve model robustness. It is resistant to over-fitting, handles both classification and regression tasks, works well with high-dimensional data, and provides feature importance scores. It can be computationally intensive and may not perform well on very small datasets and usually outperformed SVM for the test dataset [7].
- **Support Vector Machine (SVM):** The Support Vector Machine aims to find a hyperplane that best separates different classes. It is effective in both linear and non-linear scenarios and in high-dimensional spaces. It can be sensitive to the choice of kernel and hyper-parameters and does not scale well to large datasets and usually outperformed RF for the training dataset [7].
- **Decision Tree (DT):** Decision Trees partition the input space into segments based on feature values, enabling intuitive decision-making. Recursive splits based on feature importance create a tree structure [9]. It is easy to understand and interpret, handles non-linear data well, and requires minimal data pre-processing. However, it can be prone to over-fitting and can be sensitive to small variations in data.
- **k-Nearest Neighbors (k-NN):** k-Nearest Neighbors classifies data by comparing a data point with its neighbors. It classifies or labels a data point by considering the majority class of its k-nearest neighbors. The distance metric, often Euclidean distance [9], determines "nearness" as defined below:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \quad (1)$$

  Intuitive and easy to implement, it adapts well to local data characteristics. However, it can be sensitive to irrelevant features and it is computationally expensive during prediction.

2.2. Deep Learning (DL) in Memory Analysis

DL, particularly Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), has shown promise in memory analysis. CNNs can learn visual patterns, such as graphical user interface components, within memory dumps [10]. As we cannot use CNN, we will consider the data obtained from the analysis step which is in numerical format. Furthermore, DL models trained on memory dumps can automate the identification of malware signatures, the extraction of code injection attempts, and the recognition of malicious payloads embedded in memory. These models can process large amounts of memory data in real time, enabling a rapid response to emerging threats. RNN, especially LSTM (Long Short-Term Memory), models are effective in capturing sequential patterns, which is crucial for detecting complex attacks such as multi-stage malware. Refs. [7,11] have shown that the LSTM success rate/accuracy is better when compared to CNN; that is why we have selected this model for our research. A brief description of LSTM is given below:

- **Long Short-Term Memory (LSTM):** LSTM is a specialized recurrent neural network tailored for sequential data. Its unique architecture, with forget, input, and output gates, enables it to retain information and capture temporal patterns [11]. The equa-

tions given below show that the new state depends on the previous states. The new cell state is a combination of the previous cell state and the updated cell state. The output gate is calculated in the following steps.

1. Forget Gate:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (2)$$

2. Input Gate:

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \quad (3)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (4)$$

3. Cell State Update:

$$C_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_c) \quad (5)$$

4. New Cell State:

$$C_{t\text{new}} = f_t \cdot C_{t-1} + i_t \cdot C_t \quad (6)$$

5. Output Gate:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (7)$$

$$h_t = o_t \cdot \tanh(C_t) \quad (8)$$

where:

f_t : Forget gate output (0 to 1)
i_t : Input gate output (0 to 1)
$\tilde{C}_t$: Candidate cell state
C_t : Updated cell state
$C_{t\text{new}}$: New cell state
o_t : Output gate output (0 to 1)
h_t : Hidden state (output)
W_f, W_i, W_c, W_o : Weight matrices
b_f, b_i, b_c, b_o : Bias vectors

In Equation (2) The forget gate determines what information to discard from the cell state. The input gate determines in Equations (3) and (4) what new information to store in the cell state.The cell state is updated in Equation (5) using the input gate and candidate cell state. Finally, the output gate determines what part of the cell state is exposed as the hidden state in Equations (7) and (8).

In memory analysis, LSTM's ability to learn from historical sequences makes it adept at identifying complex threats such as rootkits and malware. Its capacity to remember and analyze historical memory events equips it to automate the detection of subtle anomalies and threats in memory analysis.

3. Related Work

In [12], Djenna et al. contribute to the detection of rootkits by integrating dynamic DL-based methods and heuristic approaches within a malware detection framework. By analyzing the behavior patterns and employing advanced detection techniques, their model achieved effective identification and classification of rootkits, as well as enhanced the overall capabilities of the malware detection system.

In [13] Sihwail et al. conducted a study on the effectiveness of extracting memory-based images to detect malware. They created a binary memory-based dataset available on GitHub [14] and employed classification algorithms such as SVM, RF, k-NN, Naïve Bayes (NB), and DT. Their model achieved an accuracy of 98.5% using the Volatility v2.6 framework.

Bozkir et al. [15] utilized computer vision and ML techniques to detect and classify malware by analyzing memory dumps as RGB images. Their approach improved the

detection of unknown malware by up to 20.78% across multiple ML algorithms. They employed RF, SVM, and XGBoost algorithms and demonstrated the practicality of computer vision-based schemes for protection against malicious applications.

Another study by [16] emphasized the significance of memory analysis in capturing malware footprints and extracting hidden code from obfuscated malware. The authors developed a Python-based plugin called VolMemLyzer for Volatility v2.6 that is capable of extracting 36 features and converting the results into CSV format. The plugin showed high accuracy in malware classification, and a dataset of approximately 1,900 instances was created using the tool.

Addressing the challenges of detecting obfuscated and hidden malware, Carrier et al. [17] transformed the VolMemLyzer framework to extract 26 new memory features, enhancing its efficiency [17]. The plugin was employed to detect ransomware, Trojan Horses, and spyware, and achieved an accuracy of 99% and an F1-Score of 99.02%. An extended dataset was created, contributing positively to research in this field.

In [18], the Trusted Kernel Rootkit Detection (TKRD) system combined memory forensic analysis with bio-inspired ML techniques to detect kernel rootkits. It achieved very high accuracy. In [19], a hardware-assisted Virtualization-based Kernel-level Rootkit Detection (VKRD) system was introduced. It employed ML techniques and dynamic analysis to intercept and isolate the operations of kernel modules, albeit with performance overhead. Nagy et al. [20] addressed the challenge of detecting rootkits in embedded IoT devices by utilizing dynamic analysis within Trusted Execution Environments (TEE) available in popular IoT platforms.

These studies highlight the significance of memory analysis and various detection techniques, including ML, computer vision, and dynamic analysis, in combating the threat of malware and rootkits. Collectively, these studies reveal that a multi-dimensional approach, combining memory analysis and a range of detection methodologies, holds great promise for addressing the evolving landscape of cyber threats. Table 1 presents a summary of the research discussed in this section.

Table 1. Comparison of Related Work on Rootkit Detection using Memory Analysis.

Study	Learning Algorithm	Performance Metrics	Dataset	Strengths	Limitations
Djenna et al. [12]	DNN, CNN, RF, DT	Accuracy, precision, recall, F1-score	CICAndMal2017 [21]	Utilizes dynamic deep learning and heuristic.	Lack of in-depth analysis on potential false positives. Limited exploration of feature engineering.
Sihwail et al. [13]	SVM, Naïve Bayes, k-NN, RF, DT	Accuracy, precision, recall, F1-score, False positive rates	Sihwail [14]	Utilizes memory features extracted from memory images. Incorporates feature engineering and binary vectors for training and testing.	Potential over-fitting due to high accuracy rate on training data.

Table 1. *Cont.*

Study	Learning Algorithm	Performance Metrics	Dataset	Strengths	Limitations
Bokzir et al. [15]	CNN	Accuracy, precision, recall, F1-score, ROC-AUC	Dampware10 [22]	Comprehensive dataset with malware and benign samples. Inclusion of GIST descriptors and HOG features.	Limited explanation of dataset creation process. Limited discussion on feature extraction methods.
Lashkari et al. [16]	Adaboost, RF, k-NN, DT	False positives, False negatives, Accuracy, F1-Score, Precision	VolMemLyzer [23]	Handles feature selection automatically. Resistant to over-fitting.	Could be computationally expensive.
Carrier et al. [17]	RF, DT, k-NN, Naïve Bayes, SVM, Logistic Regression	Accuracy, F1-Score, Precision, Recall	CIC-MalMem-2022 [24]	High accuracy and fast classification.	Prone to over-fitting.
Wang et al. [18]	RF, DT, Bayesian	TPR, FPR, AUC, F-measure, Accuracy	[18]	Integration of memory forensics analysis.	Limited exploration of feature selection.

4. Methodology

Figure 1 presents an overview of the steps used in this research and Figure 2 lists the tools used.

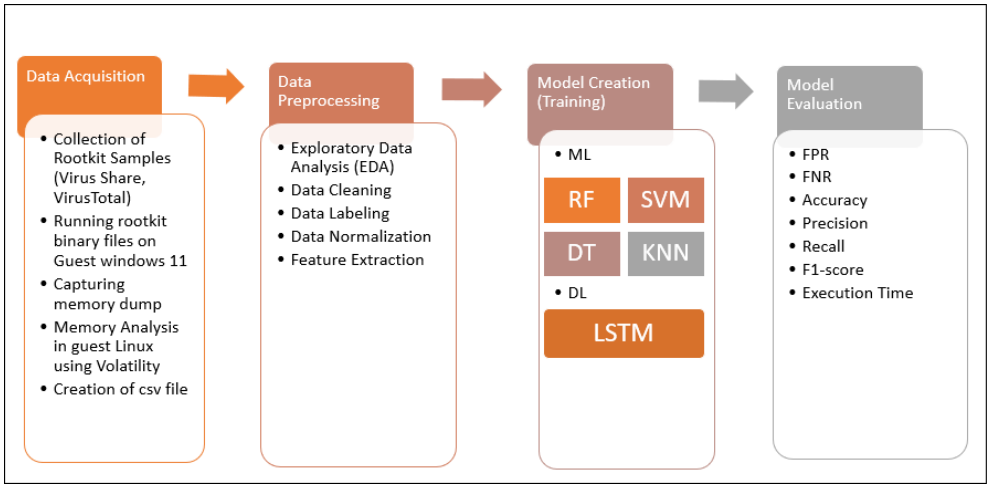

Figure 1. Overview of Methodology for Rootkit Detection using Memory Analysis.

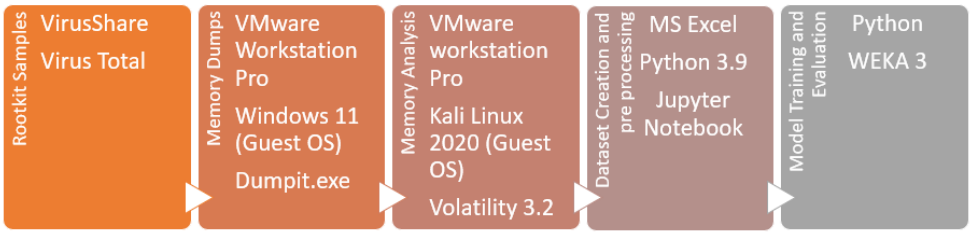

Figure 2. Tools Used for Rootkit Detection using Memory Analysis.

4.1. Data Acquisition

In this step, data are collected to build the rootkit detection model. This involves capturing the memory image (including RAM) using the Dumpit.exe tool. The generated memory dumps serve as the primary data source for further analysis.

4.1.1. Collection of Rootkit Samples

The rootkit binary files were collected from VirusShare and VirusTotal. A total of 400 binary files from various categories of rootkits (e.g., kernel rootkits, memory rootkits, etc.) were collected.

4.1.2. Creation of Memory Dump

Each rootkit sample was executed in a virtual machine running Windows 11 (guest OS) and a memory image was captured using the Dumpit.exe tool. Precautions were taken to ensure a secure environment, including Windows activation, separate network segmentation, and disabling Windows Defender. Similarly, a benign image was also captured, i.e., one without any rootkit running.

4.1.3. Feature Extraction and Memory Analysis

The Volatility v3.2 tool, installed in a Kali Linux environment, was employed for memory analysis. The memory dumps were transferred to this virtual machine for analysis. Features such as processes, DLLs, handles, modules, injected codes, networks, services, callbacks, and privileges were investigated using specific Volatility commands. The obtained information is categorized into sub-features for further analysis. Table 2 shows the features and sub-features that make up the final dataset.

- *Processes:* This involves identifying and examining the various processes running within the memory and their attributes. By analyzing the processes, it is possible to detect any suspicious or malicious programs that may be present. Rootkits often disguise themselves as legitimate processes, making the analysis of processes crucial in identifying potential threats.
- *Dynamic Link Libraries (DLLs):* DLLs are modules that contain code and data that multiple programs can use simultaneously at run time. In memory analysis, the DLLs loaded by processes are analyzed to detect any anomalous or malicious DLLs. This helps in identifying rootkit activities that involve injecting malicious code into legitimate processes through DLLs.
- *Handles:* Handles provide a way to access system resources, such as files, devices, and synchronization objects. During memory analysis, the handles utilized by processes are examined. This analysis can reveal any suspicious or unauthorized access to system resources, which can indicate the presence of rootkits.
- *Modules:* Modules refer to the executable code and associated data loaded into the memory. In memory analysis of modules, the focus is on their characteristics to identify any abnormal or unauthorized modules. Rootkits often modify modules to gain control over system processes and execute malicious activities. By analyzing the modules, potential rootkit-related modifications can be detected.
- *Injected Codes:* Injected code refers to the malicious code injected into a legitimate process's memory space. During memory analysis, this involves searching for any signs of injected code, such as unexpected modifications or additional code segments. Detecting injected code is crucial in identifying the presence of rootkits, as they often use this technique to hide their activities and evade detection.
- *Networks:* The analysis of network-related information within memory dumps helps in identifying any suspicious network connections or communications. This involves examining network-related data, such as open ports, established connections, and communication protocols. Detecting unusual or unauthorized network activities can provide insights into the presence of rootkits and their communication with external entities.
- *Services:* Services are background processes that run independently of user interactions. In memory analysis, the services running within the memory are examined to identify if they are suspicious or unauthorized. Rootkits can manipulate services to gain persistence or execute malicious activities. Analyzing services helps in detecting such manipulations and identifying potential rootkit activities.

- *Callbacks:* Callbacks are functions that are registered by processes to be executed when certain events occur. During memory analysis, registered callbacks are inspected to identify any anomalies or unauthorized changes. Rootkits can tamper with callbacks to gain control over system events and execute malicious actions. Analyzing callbacks helps in detecting such modifications and potential rootkit presence.
- *Privileges:* Privileges refer to the rights and permissions granted to processes to perform specific operations. In memory analysis, the privileges assigned to processes are examined to identify any unauthorized or elevated privileges. Rootkits often elevate privileges to gain higher system access and execute malicious activities. Analyzing privileges helps in detecting unauthorized privilege escalations and potential rootkit activities.

4.1.4. Creation of Dataset

Based on the analyzed memory dumps, a dataset is compiled by recording the values for specific features in a CSV file. The dataset includes features extracted from both benign and rootkit samples. Statistical analysis is conducted using Microsoft Excel 2019 to obtain values for sub-features, resulting in a comprehensive dataset for further analysis.

4.2. Data Pre-Processing

Data pre-processing plays a vital role in preparing the acquired data for subsequent analysis. This phase involves several steps.

4.2.1. Exploratory Data Analysis (EDA)

Typically, EDA is conducted to gain a comprehensive understanding of the dataset and its key characteristics. In this study, statistical and graphical methods were employed to explore the data and identify any patterns, anomalies, or correlations. Additionally, descriptive statistics and visualizations were utilized to gain insights into the dataset, identify patterns, and detect anomalies. EDA was conducted using Python v3.9 and libraries such as Pandas v 2.1.0 and Matplotlib v 3.7.2.

4.2.2. Data Cleaning and Labeling

Data cleaning is performed to remove inaccurate or duplicate entries from the dataset. It is also used to handle missing values by either replacing them with appropriate values or removing them from the dataset, depending on the nature and extent of the missing data. This step helps in preparing a clean dataset for further analysis. Furthermore, various techniques and methods were employed to ensure data integrity. Finally, data labeling was carried out to assign appropriate labels to the data entries, such as classifying them as **B** (benign) or **M** (malicious).

4.2.3. Data Normalization

Data normalization is applied to standardize the dataset and bring it into a defined range. This process enhances the cohesiveness of the data and facilitates subsequent analysis and modeling. By normalizing the data, the accuracy and detection rates can be improved. This process enhances the accuracy and consistency of the data. Min-max normalization is commonly used (defined in the equation below), scaling the values between 0 and 1.

$$x_{\text{normalized}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}, \tag{9}$$

Table 2. List of Features and Sub-features.

Sr No.	Features	Volatility Commands	Sub Features
1	Processes	windows.pslist	pslist.nprocc pslist.nppidd pslist.avg_threadss pslist.nprocs64bitt pslist.avg_handlerss
2	DLLs	windows.dlllist	dlllist.ndllss dlllist.avgdllsperprocc
3	Handles	windows.handles	handles.nhandless handles.avghandlesperprocc handles.nportt handles.nfilee handles.neventt handles.ndesktopp handles.nkeyy handles.nthreadd handles.ndirectoryy handles.nsemaphoree handles.ntimerr handles.nsectionn handles.nmutantt Handles.nsymboliclinkk Handles.nkeyedeventt Handles.nprocesss Handles.ntokenn Handles.nwindowstationn Handles.niocompletionn Handles.nwmiguidd handles.nwaitableportt Handles.njobb
4	Injected Codes	windows.malfind	malfind.ninjectionss malfind.commitChargee
5	Modules	windows.modules	modules.nmoduless
6	Services	windows.svcscan	svcscan.nservicess svcscan.kerneldriverss svcscan.fsdriverss svcscan.processservicess svcscan.sharedprocessservicess svcscan.interactiveprocessservicess
7	Callbacks	windows.callbacks	callbacks.ncallbackss
8	Network	windows.netscan	Netscan.nudpp Netscan.ntcpp
9	Privileges	windows.privileges	Priv.nprocesss Priv.avgprivperprocc priv.SeSystemEnvironmentPrivilegee priv.SeSystemProfilePrivilegee priv.SeSystemtimePrivilegee priv.SeTakeOwnershipPrivilegee priv.SeTcbPrivilegee priv.SeTimeZonePrivilegee priv.SeUndockPrivilege Priv.SeSecurityPrivilege

4.2.4. Feature Selection

Feature selection is a crucial step in which features that significantly impact the results are identified. This process is typically performed in conjunction with EDA and data normalization, as it helps in improving accuracy, reducing computational time, and optimizing the overall performance of the models. Recursive elimination is employed as a feature selection method, eliminating less important features from the dataset. This step improves the efficiency of the subsequent analysis and reduces computational complexity.

4.3. Model Creation (Training)

Once data pre-processing is completed, the dataset is ready for model training. In this phase, a suitable ML or DL algorithm is selected based on the research objectives and characteristics of the dataset. The selected algorithm is trained using a portion of the pre-processed dataset to learn patterns and make predictions.

For the purpose of rootkit detection, we have selected four ML algorithms, namely SVM, RF, k-NN, and DT, along with one DL algorithm called the Long- and Short-term Memory (LSTM) Model.

To train the models, the dataset is divided into two parts: 75% for training and 25% for testing. This split allows us to train the models on a significant portion of the data while reserving a separate portion for evaluating their performance.

The Algorithm 1 shows the steps for DT. Before training the model, the time is calculated, and after the model is executed, the time is again calculated, along with accuracy and other evaluation metrics. A similar method is performed for the RF, SVM, and KNN. The Algorithm 2 shows steps for LSTM.

4.4. Model Evaluation

The trained model's performance is evaluated in this phase. Testing data, separate from the training data, is used to assess the model's accuracy and generalization ability. It is crucial to ensure that the evaluation data is distinct to avoid over-fitting. If the model does not meet the required criteria, model tuning is performed by adjusting the algorithm's parameters to optimize its performance and enhance the accuracy of the results.

The evaluation process is a critical step in assessing the effectiveness of the proposed method. Several evaluation measures are used to measure the performance of the models, including the confusion matrix, accuracy, precision, recall, and F1-score. An explanation of these measures is given below:

Algorithm 1 Algorithm for DT.

1. Import necessary libraries (NumPy, Matplotlib, Pandas, Seaborn, Category Encoders, Scikit-Learn, Time).
2. Import the dataset from a CSV file.
3. Separate the dataset into features (X) and the target variable (y).
4. Split the dataset into training and testing sets.
5. Normalize the dataset using MinMaxScaler to scale values between 0 and 1.
6. Encode categorical features using an ordinal encoder.
7. Create a Decision Tree Classifier with the Entropy criterion and a maximum depth of 3.
8. Train the Decision Tree Classifier with the Entropy criterion.
9. Make predictions on the test data.
10. Calculate and print the accuracy score using the Entropy criterion.
11. Measure and print the execution time for this model.
12. Create a confusion matrix for evaluating model performance.
13. Visualize the confusion matrix using a heatmap.
14. Print a classification report for model evaluation.
15. End of the Code.

Algorithm 2 Algorithm for LSTM.

1. Import Libraries
2. Load the dataset from a CSV file
3. Label encode the 'Class' column using LabelEncoder.
4. Convert the values in the dataset to float32.
5. Split Dataset, Define `train_size` as 75% of the data and `test_size` as the remaining 25% of the data.
6. Normalize the dataset using MinMaxScaler to scale values between 0 and 1.
7. Define a function `create_dataset(dataset, look_back)` to prepare the data for LSTM:
   - Initialize empty lists `dataX` and `dataY`.
   - Loop over the dataset with a sliding window of size `look_back`.
   - Append the input sequence (`dataX`) and the corresponding output value (`dataY`).
   - Return `dataX` and `dataY`.
8. Set `look_back` to 1 (defining the sequence length).
9. Create training and testing input sequences and output values using `create_dataset`.
10. Modify the shape of `trainX` to (`train_samples`, `time_steps`, `features`).
11. Modify the shape of `testX` to (`test_samples`, `time_steps`, `features`).
12. Compile the LSTM model using 'adam' optimizer and 'mse' loss function.
13. Fit the model to training data (`trainX`, `trainY`).
14. Predict `testX` using the trained model.
15. Inverse transform the scaled predictions and true values to their original scales.
16. Calculate accuracy using accuracy_score between `testY_classes` and `testPredict_classes`.
17. End of Code.

Confusion Matrix: This is a 2×2 matrix that contains the actual and predicted values as defined below:

- True Positive (*TP*)—represents the correctly predicted positive instances;
- True Negative (*TN*)—represents the correctly predicted negative instances;
- False Positive (*FP*)—represents the incorrectly predicted positive instances;
- False Negative (*FN*)—represents the incorrectly predicted negative instances.

Accuracy: This is a measure of the overall correctness of the predictions and is calculated as the ratio of the sum of *TP* and *TN* to the total number of instances.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}, \tag{10}$$

Precision: This is the percentage of correctly predicted positive instances out of all positive predictions.

$$\text{Precision} = \frac{TP}{TP + FP}, \tag{11}$$

F1-score: This is a harmonic mean of precision and recall. It provides a balanced measure of model performance.

$$\text{F1\_score} = \frac{2 \times (Precision \times Recall)}{Precision + Recall}, \tag{12}$$

Recall: This is also known as sensitivity or true positive rate. It represents the percentage of actual positive instances that are correctly classified.

$$\text{Recall} = \frac{TP}{TP + FN}, \tag{13}$$

Finally, the *Execution Time (ET)* is another important measure that indicates the time taken by the model to process the dataset. It was calculated using the `time()` function in Python.

5. Results and Discussion

5.1. Results

This section presents the evaluation metrics for the DT, RF, SVM, k-NN, and LSTM models. The models were trained and evaluated using a 75/25 split of the dataset, with 75% used for training and 25% for testing. Feature selection reduced the initial 53 features to 47. Based on the confusion matrix, the accuracy, precision, recall, and F1-score were calculated for each model and summarized in Table 3. It is easy to see that SVM achieved the highest accuracy of 96.2%, followed by RF with an accuracy of 95.5%. LSTM, a DL model, had an accuracy of 85.8%, demonstrating its potential for rootkit detection. These results are graphically shown in Figure 3.

Table 3. Performance Comparison of Models for Rootkit Detection.

	Algorithms	Accuracy	Recall	Precision	F1-Score	Execution Time
Machine Learning	Random Forest	95.5%	93%	98%	95%	0.25 s
	K-Nearest Neighbor	92.8%	91.5%	93.8%	92.7%	**0.03 s**
	Decision Tree	95%	93%	97%	95%	0.04 s
	Support Vector Machine	**96.2%**	**94%**	**99%**	**96%**	0.1 s
Deep Learning	Long- and Short-Term Memory	85.8%	83%	87%	85%	124.02 s

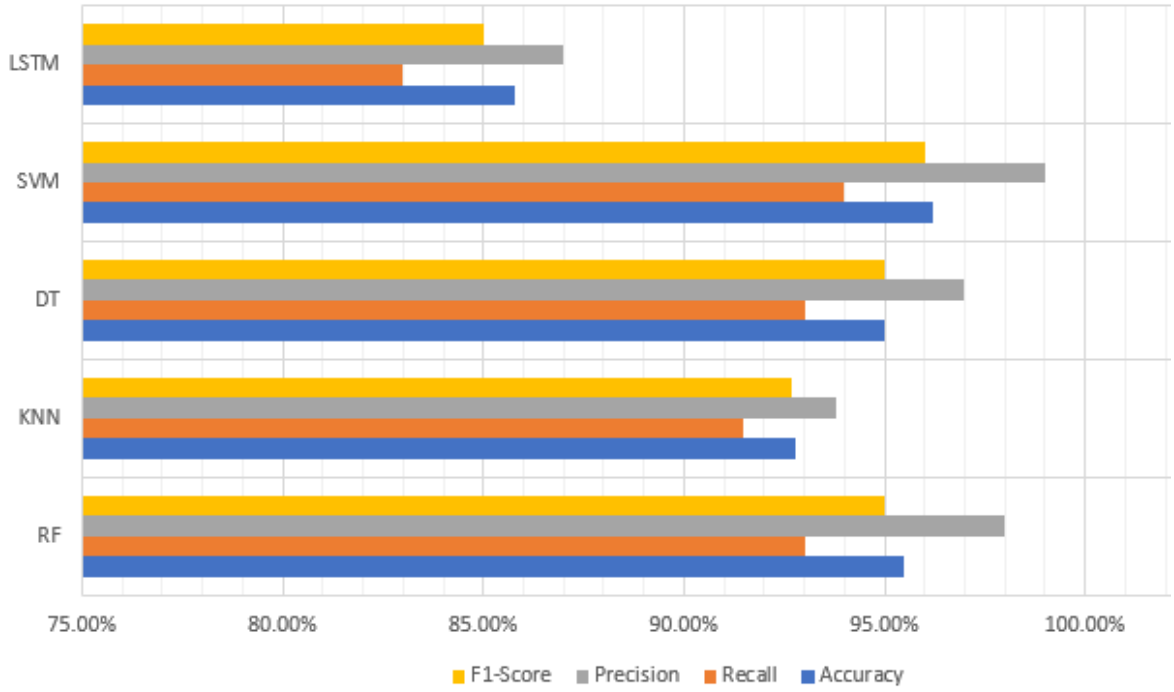

Figure 3. Comparison of Accuracy, Precision, Recall, and F1-score of Models for Rootkit Detection.

With regard to execution time, it is clear from Table 3, that k-NN provides the minimum time of 0.03 s. DT is also quite faster with an execution time of only 0.04 s. The LSTM model, on the other hand, shows the maximum execution time of 124.02 s.

Confusion matrix is also an important way to visualize the results. It is used in machine learning and classification tasks to describe the performance of a classification

model on a set of data for which the true values are known. Figure 4 shows the confusion matrices of the models.

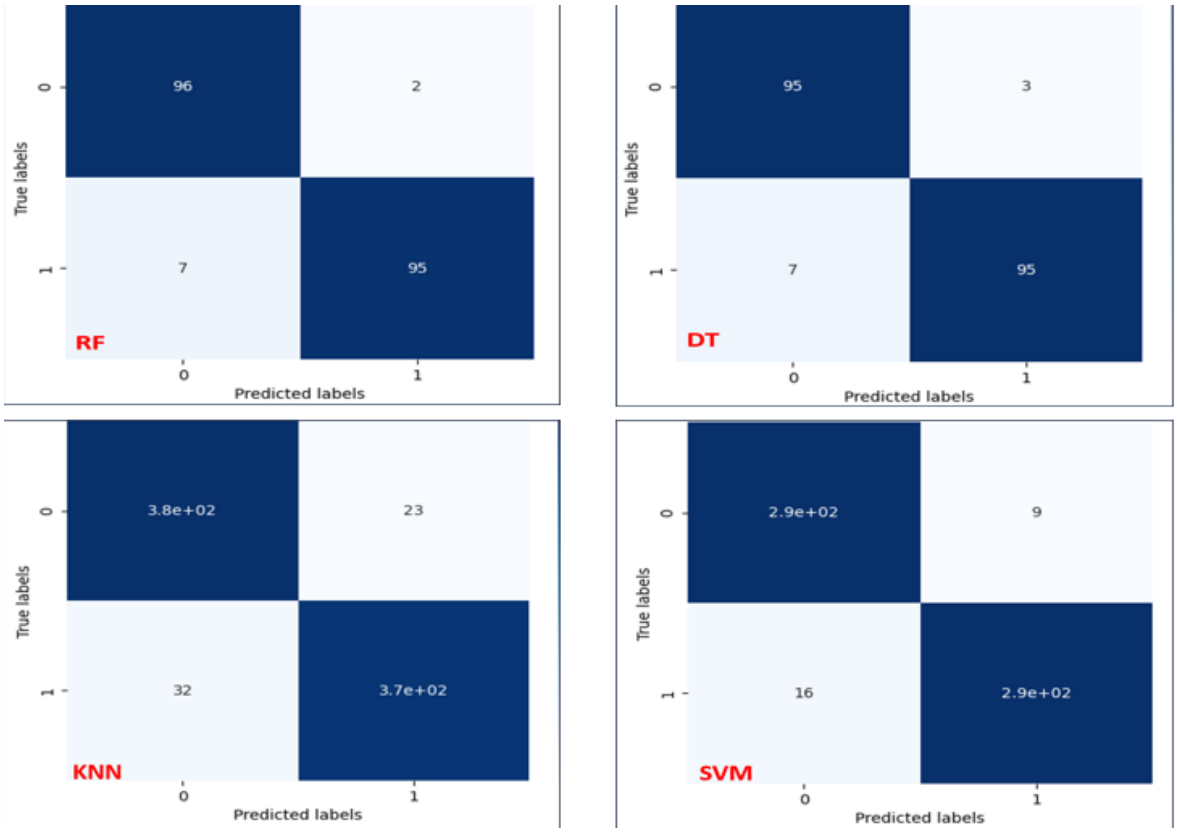

Figure 4. Confusion Matrices of Models for Rootkit Detection.

5.2. Discussion

The trained models were evaluated using metrics, such as accuracy, precision, recall, and F1-score. The SVM model showed the highest accuracy of 96.2%, followed by RF with an accuracy of 95.5%. SVM also had a shorter execution time compared to RF. K-Nearest Neighbor (k-NN) exhibited the lowest execution time among all the models. The DL model, LSTM, achieved an accuracy of 85.8%. However, LSTM had a significantly longer execution time compared to the ML classification algorithms because of the computationally intensive operations involved in DL.

One of the reasons why LSTM showed the worst results is because small datasets can pose challenges when used to trained complex models such as LSTM. Limited data often leads to over-fitting, causing the model to memorize noise rather than learn meaningful patterns. Moreover, the variance in performance on such datasets makes generalization difficult.

Table 4 shows a comparison of our model with the existing literature in terms of approches and dataset properties, whereas Table 5 shows a comparison of our model with the existing literature in terms of the accuracy of different models and execution time. Our research stands out in terms of giving importance to memory analysis for rootkit detection, utilizing Volatility 3.2 and considering a DL algorithm in addition to ML algorithms. The inclusion of execution time is also a unique aspect of our research. By addressing these gaps, our research aims to contribute to the field of rootkit detection and enhance the effectiveness of malware detection approaches.

Table 4. Comparison of Approaches and Data Sources.

Ref.	Approach	Data Source	Benign	Rootkit
[25]	Dynamic	VirusTotal	24,000	4500
[19]	Dynamic	Rootkit.com VirusShare VX Heaven	473	418
[26]	Dynamic	×	1300	700
[18]	Memory Forensics	MalShare VirusShare	2600	10,500
Our Approach	Memory Analysis	VirusTotal VirusShare VX Heaven	400	400

× = Not applicable.

Table 5. Comparison of Results (Accuracy and Execution Time).

Ref.	ML Algorithms					DL Algorithms		Execution Time
	SVM	DT	k-NN	RF	NB	FNN	LSTM	
[25]	99.9	83.71	×	×	75.35	×	×	×
[19]	×	95.11	91.85	96.74	×	×	×	×
[26]	×	×	×	×	×	67.7	×	×
[18]	94.09	96.5	×	×	×	×	×	×
Our Approach	96.2	95	92.8	95.5	×	×	85.8	✓

SVM = Support Vector Machine, DT = Decision Tree, k-NN = K nearest neighbor, RF = Random Forest, NB = Naive Bayes, FNN = Forward Neural Networks, LSTM = Long and Short Term Memory, × = Not applicable, ✓ = Performed the Metric.

There are a few more important points to note from Table 3. Firstly, in [25], the SVM model achieves 99.9% accuracy. This is higher than the accuracy of our SVM model (96.2%). It is important to note that our study uses a memory approach, whereas [25] uses a dynamic approach. Therefore, a straightforward comparison cannot be made. It is also strange that [25] has very average results with DT (83.71%) and NB (75.35%). Secondly, the only other study to use memory forensics is [18]. Our best results (96.2% for SVM) are very close to theirs (highest accuracy of 96.5% with DT). Their dataset, although larger than ours, is very unbalanced and the accuracy is very sensitive to unbalanced categories.

It is also important to point out that our DL model has a higher accuracy than the accuracy of the DL model in [26]. The FNN model in [26] has an accuracy of 67.7%, whereas our LSTM model has an accuracy of 85.8%. Moreover, we have calculated the execution time, which has not been reported in any other study on rootkit detection. The fastest time is reported by k-NN and the slowest by the DL model LSTM.

6. Conclusions and Future Work

The insidious nature of rootkits poses a significant threat to system security and data integrity. The existing approaches for rootkit detection have limitations, and there is a need for an efficient and automated mechanism to identify these malicious programs.

In this study, the potential approach for rootkit detection using memory analysis and ML/DL algorithms was explored. The research methodology involved data acquisition, data pre-processing, model creation, and model evaluation. Memory dumps were captured using Dumpit.exe and then analyzed using volatility. Relevant features were extracted, collected, pre-processed, and compiled into a dataset that was subsequently used to train different ML and DL models.

The results demonstrated the efficiency of the proposed approach in accurately detecting rootkits. SVM emerged as the most effective model, achieving the highest accuracy

rate of 96.2%. K-NN was the fastest model, with a minimum execution time of 0.03 s. Our study is also the first to report the execution time of rootkit detection models. Furthermore, our DL model has almost 18% higher accuracy than the accuracy of the DL model reported in the existing literature.

The findings of this research contribute to the field of rootkit detection by showcasing the potential of memory analysis combined with ML and DL algorithms. By leveraging memory analysis, the proposed approach can detect rootkits that may go undetected by traditional detection tools. The incorporation of ML and DL algorithms further enhances the accuracy and efficiency of rootkit detection.

This research opens avenues for further exploration and improvement in rootkit detection techniques. Future work can focus on refining the models, exploring additional features, expanding the dataset to enhance the overall performance of the detection system, the family classification of rootkits, and reducing the time of memory analysis. The proposed approach can be integrated into existing security systems to bolster their capabilities in identifying and mitigating rootkit threats, thereby enhancing the security of computer systems and protecting against potential data breaches. However, there are some limitations too. The small dataset creates overhead and causes the DL to get low accuracy values.

Author Contributions: Conceptualization, S.Q. and B.N.; formal analysis, B.N.; supervision, S.Q.; investigation, B.N.; validation, B.N. and S.Q.; visualization, B.N.; writing—original draft, B.N.; writing—review and editing, S.Q. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data can be obtained from the corresponding author upon reasonable request.

Acknowledgments: We would like to express our sincere gratitude to the School of Electrical Engineering and Computer Science for their invaluable support and provision of extra memory resources. The additional memory, made available through the use of an HDD (Hard Disk Drive), has been instrumental in enhancing our research and academic endeavors. We also ensure that all individuals included in this section have consented to the acknowledgement

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Manap, S. Rootkit: Attacker Undercover Tools. Available online: http://forum.ouah.org/salirootkit.pdf (accessed on 6 April 2020).
2. Bickford, J.; O'Hare, R.; Baliga, A.; Ganapathy, V.; Iftode, L. Rootkits on smartphones. In Proceedings of the Eleventh Workshop on Mobile Computing Systems Applications-HotMobile '10, Annapolis, MD, USA, 22–23 February 2010. [CrossRef]
3. Bunten, A. UNIX and Linux based Rootkits Techniques and Countermeasures. 2004. Available online: https://www.semanticscholar.org/paper/UNIXand-Linux-based-Rootkits-Techniques-and-Bunten/ (accessed on 31 March 2023).
4. Huseynov, H.; Saadawi, T.; Kourai, K. Hardening the Security of Multi-Access Edge Computing through Bio-Inspired VM Introspection. *Big Data Cogn. Comput.* **2021**, *5*, 52. [CrossRef]
5. Koushki, M.M.; AbuAlhaol, I.; Raju, A.D.; Zhou, Y.; Giagone, R.S.; Shengqiang, H. On building machine learning pipelines for Android malware detection: A procedural survey of practices, challenges and opportunities. *Cybersecurity* **2022**, *5*, 16. [CrossRef]
6. Gibert, D.; Mateu, C.; Planes, J. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *J. Netw. Comput. Appl.* **2020**, *153*, 102526. [CrossRef]
7. Halbouni, A.; Gunawan, T.S.; Habaebi, M.H.; Halbouni, M.; Kartiwi, M.; Ahmad, R. Machine Learning and Deep Learning Approaches for CyberSecurity: A Review. *IEEE Access* **2022**, *10*, 19572–19585. [CrossRef]
8. Vashishtha, L.K.; Chatterjee, K.; Rout, S.S. An Ensemble approach for advance malware memory analysis using Image classification techniques. *J. Inf. Secur. Appl.* **2023**, *77*, 103561. [CrossRef]
9. Xin, Y.; Kong, L.; Liu, Z.; Chen, Y.; Li, Y.; Zhu, H.; Gao, M.; Hou, H.; Wang, C. Machine Learning and Deep Learning Methods for Cybersecurity. *IEEE Access* **2018**, *6*, 35365–35381. [CrossRef]
10. Aksan, F.; Li, Y.; Suresh, V.; Janik, P. CNN-LSTM vs. LSTM-CNN to Predict Power Flow Direction: A Case Study of the High-Voltage Subnet of Northeast Germany. *Sensors* **2023**, *23*, 901. [CrossRef] [PubMed]
11. Aydın, H.; Orman, Z.; Aydın, M.A. A long short-term memory (LSTM)-based distributed denial of service (DDoS) detection and defense system design in public cloud network environment. *Comput. Secur.* **2022**, *118*, 102725. [CrossRef]

12. Djenna, A.; Bouridane, A.; Rubab, S.; Marou, I.M. Artificial Intelligence-Based Malware Detection, Analysis, and Mitigation. *Symmetry* **2023**, *15*, 677. [CrossRef]
13. Sihwail, R.; Omar, K.; Ariffin, K.A.Z. An Effective Memory Analysis for Malware Detection and Classification. *Comput. Mater. Contin.* **2021**, *67*, 2301–2320. [CrossRef]
14. Sihwail. Sihwail/Malware-Memory-Dataset. GitHub. 5 February 2021. Available online: https://github.com/sihwail/malware-memory-dataset (accessed on 14 April 2023).
15. Bozkir, A.S.; Tahillioglu, E.; Aydos, M.; Kara, I. Catch them alive: A malware detection approach through memory forensics, manifold learning and computer vision. *Comput. Secur.* **2021**, *103*, 102166. [CrossRef]
16. Lashkari, A.H.; Li, B.; Carrier, T.L.; Kaur, G. VolMemLyzer: Volatile Memory Analyzer for Malware Classification using Feature Engineering. In Proceedings of the 2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS), Hamilton, ON, Canada, 18–19 May 2021. Available online: https://ieeexplore.ieee.org/document/9452028 (accessed on 23 April 2023)
17. Carrier, T.; Victor, P.; Tekeoglu, A.; Lashkari, A. Detecting Obfuscated Malware using Memory Feature Engineering. In Proceedings of the 8th International Conference on Information Systems Security and Privacy, Online, 9–11 February 2022. [CrossRef]
18. Wang, X.; Zhang, J.; Zhang, A.; Ren, J. TKRD: Trusted kernel rootkit detection for cybersecurity of VMs based on Machine Learning and memory forensic analysis. *Math. Biosci. Eng.* **2019**, *16*, 2650–2667. [CrossRef] [PubMed]
19. Tian, D.; Ma, R.; Jia, X.; Hu, C. A Kernel Rootkit Detection Approach Based on Virtualization and Machine Learning. *IEEE Access* **2019**, *7*, 91657–91666. [CrossRef]
20. Nagy, R.; Németh, K.; Papp, D.; Buttyán, L. Rootkit Detection on Embedded IoT Devices. *Acta Cybern.* **2021**, *25*, 369–400. [CrossRef]
21. Lashkari, A.H.; Kadir, A.F.A.; Taheri, L.; Ghorbani, A.A. Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification. In Proceedings of the 2018 International Carnahan Conference on Security Technology (ICCST), Montreal, QC, Canada, 22–25 October 2018; pp. 1–7. [CrossRef]
22. Dumpware 10 Dataset Homepage. Available online: https://web.cs.hacettepe.edu.tr/~selman/dumpware10/ (accessed on 17 August 2023).
23. Volatility Memory Analyzer. 2019. Available online: https://github.com/ahlashkari/VolMemLyzer (accessed on 23 April 2023)
24. CIC-MalMem-2022. 2022. Available online: https://www.unb.ca/cic/datasets/malmem-2022.html (accessed on 23 April 2023)
25. Singh, B.; Evtyushkin, D.; Elwell, J.; Riley, R.; Cervesato, I. On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017. [CrossRef]
26. Luckett, P.; McDonald, J.T.; Dawson, J. Neural Network Analysis of System Call Timing for Rootkit Detection. In Proceedings of the 2016 Cybersecurity Symposium (CYBERSEC), Coeur d'Alene, ID, USA, 18–20 April 2016. Available online: https://ieeexplore.ieee.org/abstract/document/7942417 (accessed on 23 April 2022).