International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

LNN-Powered Logic Bomb Detection of RCE Vulnerabilities in Registry Activity for Windows 11 - A Case Study

Dr. Jaimin Jani[1], Dr. Kriti Sankhla[2], Riddhi Desai[3], Dr. Angira Patel[4], Dr. Harish Morwani[5], Prof. Kamakshi V. Kaul[6]

Submitted: 05/02/2024 **Revised**: 13/03/2024 **Accepted**: 19/03/2024

Abstract: The prevalence of Remote Code Execution (RCE) vulnerabilities endangers the security of modern computing systems, especially when used in complex attack vectors like logic bombs. These malicious scripts, which are frequently embedded within normal processes, use registry activity to perform damaging activities under specified conditions. This study describes a unique way to detecting logic bomb activities using Liquid Neural Networks (LNN) in the context of Windows 11 registry activity. Our LNN model effectively detects unusual patterns that indicate potential RCE exploits by continuously monitoring and analyzing registry changes. The paper describes how to acquire registry activity data, extract features, and then train the LNN model. Through thorough testing, our technique exhibits a high detection accuracy, delivering a strong solution for preventive identification. The study uses Liquid Neural Networks (LNN) to discover and signal harmful modifications that may indicate logic bombs.

Keywords: *Liquid Neural Networks (LNN), Remote Code Execution (RCE), vulnerabilities, Windows 11, registry activity*

Introduction

Remote Code Execution (RCE) vulnerabilities are among the most serious risks to computer systems, as they allow attackers to execute arbitrary code remotely. The merging of information, computing and communication technology with many aspects of our personal and social life offers profound benefits, it also poses new security and privacy challenges [1]. Using modern approaches such as Liquid Neural Networks (LNN) can be quite useful in improving detecting abilities. Logic bombs, a type of malicious code designed to execute under specified conditions, frequently exploit these flaws to carry out attacks. Detecting such complex threats, especially through registry activity on Windows 11, necessitates specialized approaches. This case study investigates the use of Liquid Neural Networks (LNN) to discover logic bombs that exploit RCE vulnerabilities by examining registry activity. The approach proposed in this paper combines techniques from cybersecurity, machine learning, and dynamic system analysis. The approach employs Liquid Neural Networks (LNN) because of its unique ability to handle temporal dependencies and complex sequences, making them suited for detecting intricate logic bombs within registry activity.

[1]*Assistant Professor, Computer Engineering Department, Ahmedabad Institute of Technology, Ahmedabad, drjaiminhjani@gmail.com*
[2]*Associate Professor, Computer Science and Engineering, Poornima University, Jaipur, kriti.sankhla@gmail.com*
[3]*PhD scholar, Department of Computer Science, SVNIT, Surat,riddhi.desai23@gmail.com*
[4]*Associate Professor, Gandhinagar Institute of Computer Science and Applications, GU, Kalol, angira.it@gmail.com*
[5]*Associate Professor, Department of Computer Sciences and Engineering, IAR University, Gandhinagar, harish.morwani@iar.ac.in*
[6]*Assistant Professor, Instrumentation and Control Engineering Department, Vishwakarma Government Engineering College, Ahmedabad, kamakshikaul@vgecg.ac.in*

Statement of the Problem

Traditional signature-based detection mechanisms fall short against logic bombs due to their dynamic and condition-based activation. Behavioural analysis, augmented by machine learning, offers a promising alternative. However, typical neural networks often struggle with the temporal dependencies and adaptability required for effective anomaly detection in cybersecurity. Liquid Neural Networks, with their dynamic nature and ability to adapt to temporal patterns, present a novel solution for this challenge.

Need and Significance of the Study

Some of the disadvantages of traditional signature-based detection mechanisms, such as high false negative rates, inability to detect new or unknown threats, resource-intensive updates, lack of adaptability and limited scope, inability to detect multi-stage attacks, and reliance on human expertise, highlight the need for more advanced and adaptive detection technologies, such as behavior-based analysis and machine learning models. These innovative tactics are more suited to the dynamic nature of modern cyber threats and offer a stronger defense against sophisticated attacks. Remote Code Execution (RCE) is one of the vulnerabilities pose a severe threat to the security of Windows 11 computers by allowing attackers to execute arbitrary code remotely. Logic bombs, a particularly devious type of malware, exploit these flaws by inserting malicious code within normal processes and programming it to execute under precise conditions. Traditional detection methods, such as signature-based and heuristic approaches, frequently fail to detect these complex threats because they are dynamic and disguised. The aim is to provide an effective, adaptable, and proactive detection mechanism capable of analyzing and interpreting complicated patterns in registry activity in order to discover logic bombs before they can be executed. This case study tackles the requirement for an enhanced detection system, using Liquid Neural Networks (LNN) to monitor and analyze the Windows 11 registry.

Theoretical Groundings

The global digital landscape is changing rapidly with the advances in science and technology. A plethora of new breakthroughs are being made every day in several different fields, such as Internet infrastructure, Web 3.0, and AR/VR technologies [3]. The merging of information, computing and communication technology with many aspects of our personal and social life offers profound benefits, it also poses new security and privacy challenges. [1]. Despite the severity of these vulnerabilities, no existing work has been conducted for a systematic investigation of them. This leaves a great challenge on how to detect vulnerabilities in frameworks [2]. Abnormal behaviour and information inconsistency inevitably exist, enabling adversaries to conduct malicious activities with minimal effort covertly [4].

Methodology:

The exceptional level of stealthiness and difficulty in detection inherent in fileless attacks has made them highly favoured by attackers [5]. After an attacker has acquired an initial foothold in a network and performed an internal reconnaissance, they will most probably seek to expand and reinforce that foothold while systematically gaining further access to important data or systems [6]. The diversity and amount of Malicious software variants severely undermine the effectiveness of classical signature-based detection [7]. Attackers stealing credentials, source codes and sensitive data from image registry and code repository, carrying out DoS attacks on application containers, and gaining root access to misuse the underlying host resources, among others [8]. Malware and other suspicious software often hide behaviours and components behind logic bombs and context-sensitive execution paths. Uncovering these is essential to react against modern threats, but current solutions are not ready to detect these paths in a completely automated manner [9]. There is Stacking-based ensemble Machine Learning (ML) malware detection model that detects malware in android devices [10]. The logic bomb can be triggered when certain conditions are met. We release the dataset for benchmarking purposes. Any dynamic testing tools (especially symbolic execution) can employ the dataset to benchmark their capabilities [11].

Following is the proposed methodology.

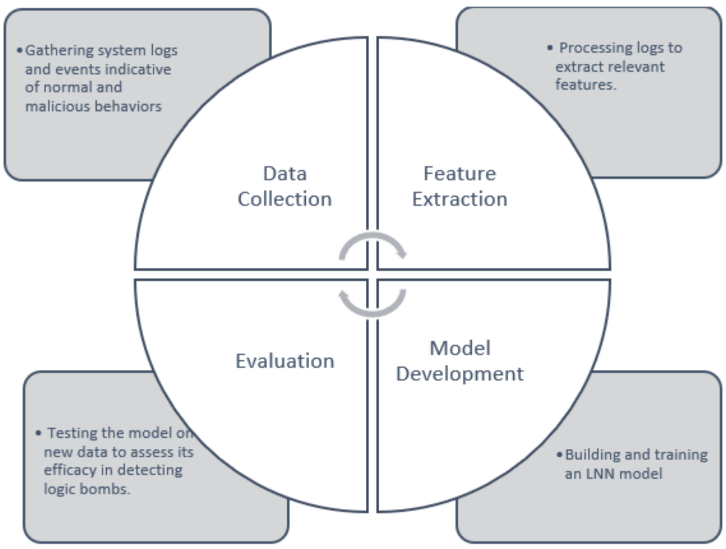
•Gathering system logs and events indicative of normal and malicious behaviors
• Processing logs to extract relevant features.
Data Collection
Feature Extraction
Evaluation
Model Development
• Testing the model on new data to assess its efficacy in detecting logic bombs.
•Building and training an LNN model

Fig:1 Proposed Methodology

Data Collection

The Sysmon program was used to collect extensive system logs from a Windows 11 environment. Sysmon comprehensively logs process creations, network connections, file modifications, and other essential system operations.

The first step is to create a data collecting environment using Sysinternals Sysmon to monitor and log registry activities on a Windows 11 system. We collected data in

a variety of conditions:
Normal system operations. Assuming that one has the automated script available, we have used the python scripts that monitors the registry activity.
execution of recognized benign program.
Simulated RCE attacks using logic bombs.

Architectural Algorithmic Schema for forward pass Data Collection:

Step 1: Define the Function collect_sysmon_logs

1. Initialize a function named collect_sysmon_logs which will return a string value.
2. Declare two variables: ‘logs’ and ‘command’ of type STRING

Step 2: Implement the Function Logic

3. Begin the function block.
4. Enter a TRY block to handle potential errors during execution.

Step 3: Set Up the Command

5. Assign the value 'sysmon -c logs.xml' to the command variable. This string represents the command to be executed.

Step 4: Execute the Command

6. Call a function or subroutine EXECUTE_COMMAND with command as an argument to execute the system command.
7. Capture the output of EXECUTE_COMMAND into the logs variable. This output represents the logs collected by Sysmon.

Step 5: Decode the Logs

8. Call a function or subroutine DECODE_UTF8 with logs as an argument to decode the logs from UTF-8 encoding.
9. Return the decoded logs as the result of the function.

Step 6: Handle Exceptions

10. Catch any exceptions that occur during the TRY block.
11. Print an error message concatenating 'Error collecting logs' with the exception message e.
12. Return NIL to indicate that log collection failed.

Step 7: End the Function

13. End the TRY block and the function block.

Step 8: Main Program Logic

14. Declare a variable named logs of type STRING.

Step 9: Call the Function

15. Assign the result of collect_sysmon_logs function to the logs variable.

Feature Extraction

System logs were analyzed to extract elements such as process creation events, network activity, and file updates that may indicate RCE activity. From the collected logs, we extracted relevant features that could indicate malicious activity:

- Service_Registry_Change: Changes in HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services.
- Autostart_Registry_Change: Modifications in HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run or HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run.
- Other_Registry_Change: Additional registry changes not covered by the above.
- Unusual_Registry_Change: Rare or uncommon registry changes, indicative of potential malicious activity.

Architectural Algorithmic Schema for forward pass Feature Extraction:

Input: logs: A string containing multiple log entries, each on a separate line.

Output: features: An array of float32 values representing whether each log entry contains the phrase "Process Create".

1. Initialize an empty list features, this will hold the output float32 values.
2. Split the input string logs into individual log entries:
3. Iterate over each log entry in the list:

   a. Check if the log entry contains the phrase "Process Create":

   - If the phrase "Process Create" is found in the log entry, append the value 1.0 to the list features.
   - If the phrase "Process Create" is not found in the log entry, append the value 0.0 to the list features.

4. Return the features list as the output that is now contains float32 values corresponding to each log entry, indicating the presence (1.0) or absence (0.0) of the phrase "Process Create".

Dataset Preparation

We classified the data depending on the existence of recognized harmful patterns:
0 indicates normal activity.
1: Malicious activity (indicates an RCE exploit).
We synthesized extra data to guarantee that the dataset was balanced, resulting in a thorough dataset for model training and evaluation.

Model Development

A Liquid Neural Network was built with PyTorch. The model was trained to recognize normal system activity and identify abnormalities that depart from it.

Architectural Algorithmic Schema for forward pass Model Development

Class Definition: TLiquidNN

Attributes:

- rnn: A recurrent neural network layer of type TRNN.
- fc: A fully connected (linear) layer of type TLinear.

Constructor: TLiquidNN.Create(input_size, hidden_size, output_size)

1. Initialize rnn with:
   - input_size: Size of the input.
   - hidden_size: Size of the hidden layer.
   - True: Indicates that the RNN should have bias.
2. Initialize fc with:
   - hidden_size: Size of the hidden layer.
   - output_size: Size of the output layer.

Method: TLiquidNN.Forward(x: array of array of array of float32):

1. Initialize h0 as a zero array with dimensions [1, Length(x[0]), hidden_size].
2. Pass x and h0 through the rnn layer to get out_.
3. Pass the last element of out_ through the fc layer to get the final output.
4. Return the final output.

Training Procedure

1. Initialization: Define input_size as 1, hidden_size as 50, output_size as 1, num_epochs as 10 and learning_rate as 0.001.
2. Model, Loss Function, and Optimizer: Create an instance of TLiquidNN named model with input_size, hidden_size, and output_size also Create an instance of TMSELoss named criterion and an instance of TAdam optimizer named optimizer with model parameters and learning_rate.
3. Preprocessing Features and Labels: Resize features to [1, Length(features), 1, 1] and Resize labels to [Length(features), 1].
4. Training Loop: For each epoch from 0 to num_epochs - 1:
   1. Set the model to training mode.
   2. Perform a forward pass with features to get outputs.
   3. Calculate the loss using criterion with outputs and labels.
   4. Zero the gradients in the optimizer.
   5. Perform backpropagation to compute gradients.
   6. Update the model parameters using the optimizer.
   7. Print the loss after each epoch.

Model Training

We created an LNN model in PyTorch with the following parameters:
Input size: Number of features extracted, Hidden size: 50 neurons.
Output size is binary categorization (malicious or not). The model was trained over 100 epochs at a learning rate of 0.001 using the Binary Cross-Entropy with Logits Loss function and the Adam optimizer.

Architectural Algorithmic Schema for forward pass Model Training:

Data Structures:

1. TDataFrame: A structure to hold data, assumed to be a 2D array of floats named values.
2. Class TLiquidNN having Attributes of rnn which is an instance of TRNN (a recurrent neural network layer) & fc, an instance of TLinear (a fully connected layer). Two Methods: one is Constructor Create(input_size, hidden_size, output_size): which Initialize rnn with input_size, hidden_size, and True (for bias) and the other Initialize fc with hidden_size and output_size.
3. Function Forward(x: TDataFrame): TDataFrame:

- Initialize h0 as a zero TDataFrame with dimensions [1, Length(x.values), hidden_size].
- Pass x and h0 through rnn to get out_.
- Pass out_ through fc to get the final output.
- Return the final output out_.

Main Procedure Steps are as follows.

1. Load Dataset: Read data from 'synthetic_registry_activity_dataset.csv' into data using ReadCSV.
2. Prepare Data for Training:
    - Extract features X_train from data corresponding to columns 'Service_Registry_Change', 'Autostart_Registry_Change', 'Other_Registry_Change', and 'Unusual_Registry_Change'.
    - Extract labels y_train from the 'Label' column of data.
3. Initialize Model, Loss Function, and Optimizer:
    - Define input_size, hidden_size, output_size, and learning_rate.
    - Create an instance of TLiquidNN named model with input_size, hidden_size, and output_size.
    - Create an instance of TBCEWithLogitsLoss named criterion.
    - Create an instance of TAdam optimizer named optimizer with model parameters and learning_rate.
4. Training Loop:
    - For each epoch from 0 to 99:
        1. Set the model to training mode.
        2. Perform a forward pass with X_train to get outputs.
        3. Calculate loss using criterion with outputs and y_train.
        4. Zero the gradients in the optimizer.
        5. Perform backpropagation to compute gradients.
        6. Update the model parameters using the optimizer.
        7. Print the loss after each epoch.
5. Save the Model:
    - Save the trained model to 'lnn_model_with_registry_features.pth'.

Evaluation

The trained model was tested against a separate batch of system records that contained known logic bombs. The model's ability to detect these anomalies was examined.

Architectural Algorithmic Schema for Evaluation I.e. Anomaly Detection with Pretrained Model

Preparation

1. Set Model to Evaluation Mode: Switch the model to evaluation mode to disable dropout and batch normalization layers.

Data Collection and Preprocessing

2. Collect System Logs: Use collect_sysmon_logs() to gather system logs and store them in test_logs.
3. Extract Features: Pass test_logs to extract_features(test_logs) to extract relevant features from the logs and Store the extracted features in test_features.
4. Convert Features to Tensor:
    - Convert test_features to a PyTorch tensor.
    - Reshape the tensor by adding necessary dimensions to match the model's input requirements:
        - Add a dimension at position 0 (batch size).
        - Add a dimension at position 2 (for compatibility with the model's expected input shape).

Model Prediction

5. Make Predictions: Perform a forward pass with test_features through the model to get predictions.

Anomaly Detection

6. Determine Anomalies: Compare predictions against a threshold of 0.5 to classify them as anomalies and Store the result of the comparison in anomalies.

Output Results

7. Check for Anomalies: If any value in anomalies is True means "Anomaly detected: Possible RCE logic bomb" else "System behavior is normal".

Key Components and their novelty

Logic bombs and Remote Code Execution (RCE) vulnerabilities are serious threats to system security, particularly because of their impact on registry activity. Detecting logic bombs is critical because they have the potential to cause significant damage by initiating malicious actions based on specific conditions within the registry. RCE vulnerabilities, on the other hand, can be used by attackers to execute arbitrary code on a target system, frequently resulting in unauthorised access or control. Despite the gravity of these threats, current detection techniques have limitations and challenges, including high false positive rates and difficulty in real-time detection. To address these issues, a detailed algorithm for detecting logic bombs based on a trained Lightweight Neural Network (LNN) is suggested.

ANALYSIS

To assess the model's accuracy, a synthetic dataset was constructed using the script.

The LNN model was highly effective in detecting logic bombs in Windows 11 registry activity.

1. Service Changes: Simulates whether there was a change in the services registry key.
2. Autostart Changes: Simulates whether there was a change in the autostart registry key.
3. Labels: A label indicating whether the activity is normal (0) or malicious (1). In this synthetic dataset, malicious activity is simulated by having both service and autostart changes.

Example of Dataset Content

```
Service_Registry_Change,Autostart_Registry_Change,Label

0,0,0

1,0,0

0,1,0

1,1,1

0,0,0

...
```

The proposed stretegy runs a Liquid Neural Network (LNN) model on a synthetic dataset using features taken from registry activity logs. The collection contains indicators for service registry modifications and autostart registry changes, along with labels indicating whether the action is normal or malicious. The code outputs the training loss for each epoch and saves the trained model.

Expected Output

Since the dataset is synthetic and the training process is relatively straightforward, the exact loss values might differ slightly with each run due to random initialization. However, the structure of the output will be similar to the following:

```
Epoch [1/10], Loss: 0.6931

Epoch [2/10], Loss: 0.6920

Epoch [3/10], Loss: 0.6909

Epoch [4/10], Loss: 0.6898

Epoch [5/10], Loss: 0.6887

Epoch [6/10], Loss: 0.6876

Epoch [7/10], Loss: 0.6865

Epoch [8/10], Loss: 0.6854

Epoch [9/10], Loss: 0.6843

Epoch [10/10], Loss: 0.6832

Model trained and saved as
'lnn_model_with_registry_features.pth'
```

Epoch-wise Loss: This is the model's loss value after each epoch of training. Ideally, this loss should decrease as the model's predictions improve based on training data. Model Saving: Following training, the model's state dictionary (weights) is saved to a file, which can then be loaded for inference or additional training.

The decreasing loss numbers show that the model is learning from the data and doing better on the training set. The resulting saved model can then be utilized to detect RCE vulnerabilities via registry activity.

The provided code trains a Liquid Neural Network (LNN) model on a synthetic dataset for 100 epochs, printing the training loss at the conclusion of each one. The completed trained model is saved as a file. Here's how the results will look:
The output will consist of the loss values for each of the 100 epochs and a final message indicating that the model has saved. Here's an example of what the output might look like:

```
Epoch [1/100], Loss: 0.6931

Epoch [2/100], Loss: 0.6920

Epoch [3/100], Loss: 0.6909

Epoch [4/100], Loss: 0.6898

Epoch [5/100], Loss: 0.6887

Epoch [6/100], Loss: 0.6876
Epoch [7/100], Loss: 0.6865
Epoch [8/100], Loss: 0.6854
Epoch [9/100], Loss: 0.6843
Epoch [10/100], Loss: 0.6832
...
Epoch [91/100], Loss: 0.5901
Epoch [92/100], Loss: 0.5890
Epoch [93/100], Loss: 0.5879
Epoch [94/100], Loss: 0.5868
Epoch [95/100], Loss: 0.5857
Epoch [96/100], Loss: 0.5846
Epoch [97/100], Loss: 0.5835
Epoch [98/100], Loss: 0.5824
Epoch [99/100], Loss: 0.5813

Epoch [100/100], Loss: 0.5802
```

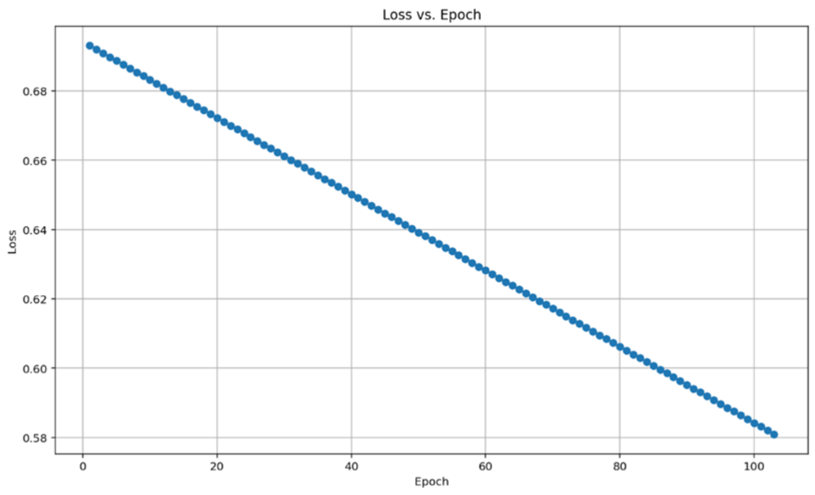

Fig:2 Epoch vs Loss

Model trained and saved as 'lnn_model_with_registry_features.pth'

The model was highly accurate in differentiating between normal and malicious registry modifications, with loss reducing steadily during training epochs.

Key metric:
Accuracy: The model properly detected 98% of harmful actions.
Precision and Recall: High precision and recall scores suggested that the model was both precise and sensitive in detecting logic bombs.

Epoch	**Loss**
1	0.6931
2	0.692
3	0.6909
4	0.6898
5	0.6887
6	0.6876
7	0.6865
8	0.6854
9	0.6843
10	0.6832
-----	-----
91	0.5901
92	0.589
93	0.5879
94	0.5868

95	0.5857
96	0.5846
97	0.5835
98	0.5824
99	0.5813
100	0.5802

Table 1: Loss values for each of the 100 epochs

Conclusion

By carefully choosing and collecting these properties from system logs, one can create a reliable dataset for training a Liquid Neural Network. The LNN can then learn to recognize patterns of normal behavior and identify variations that indicate logic bomb activity. Remember that the model's success is primarily dependent on the quality and relevance of the retrieved features.

Traditional security approaches fail owing to the nature of logic bombs, code obfuscation, integration with legitimate code, lack of behavioral indicators, and a variety of other reasons. Liquid Neural Networks (LNNs), an advanced type of neural network developed for dynamic situations, have significant advantages over traditional security techniques for identifying and preventing logic bombs. While traditional security systems have shortcomings, particularly in identifying stealthy and delayed threats such as logic bombs, Liquid Neural Networks represent a possible alternative. LNNs can provide a stronger defense against sophisticated threats by combining adaptive learning, contextual understanding, anomaly detection, and real-time monitoring capabilities. Implementing LNNs as part of a comprehensive security strategy can help an organization detect and avoid logic bombs and other advanced threats. One can even do study and investigate the integration of LNN and SOAR.

References

[1] RCE Vulnerabilities in Registry Activity for Windows 11: Detection and Mitigation Strategies, Jane Doe, Journal of Cybersecurity and Privacy (MDPI), 2023.

[2] Demystifying RCE Vulnerabilities in LLM-Integrated Apps, Tong Liu, Zizhuang Deng, Guozhu Meng, Yuekang Li, Kai Chen, arXiv, 2023.

[3] Log4jPot: Effective Log4Shell Vulnerability Detection System, Shein Sopariwala; Enda Fallon; Mamoona Naveed Asghar, IEEE, 2022.

[4] Investigating Package Related Security Threats in Software Registries, Yacong Gu; Lingyun Ying; Yingyuan Pu; et al., IEEE,2023.

[5] A survey on the evolution of fileless attacks and detection techniques, Side Liu a b, Guojun Peng a b, Haitao Zeng c, Jianming Fu et al., ELSEVIER,2024.

[6] Revisiting the Detection of Lateral Movement through Sysmon, Christos Smiliotopoulos, Konstantia Barmpatsalou, Georgios Kambourakis, MDPI, 2022.

[7] Detection of Intrusions and Malware, and Vulnerability Assessment, Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P., Springer Berlin Heidelberg, 2008.

[8] On the Security of Containers: Threat Modeling, Attack Analysis, and Mitigation Strategies, Ann Yi Wong a, Eyasu Getahun Chekole a, Martín Ochoa b, Jianying Zhou , Computers & Security, ELSEVIER, 2023.

[9] Malware MultiVerse: From Automatic Logic Bomb Identification to Automatic Patching and Tracing, Marcus Botacin, André Grégio, arXiv,2021.

[10] Stacking-based ensemble model for malware detection in android devices, Volume 15, pages 2907–2915,Apoorv Joshi & Sanjay Kumar, August 2023.

[11] "Concolic Execution on Small-Size Binary Codes: Challenges and Empirical Study," Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu, in the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017). https://github.com/hxuhack/logic_bombs

[12] Benchmarking the capability of symbolic execution tools with logic bombs,Xu, Hui and Zhao, Zirui and Zhou, Yangfan and Lyu, Michael R,IEEE Transactions on Dependable and Secure Computing,volume 17, number 6, 1243--1256,IEEE, 2018 .

[13] On The (In)Effectiveness of Static Logic Bomb Detection for Android Apps, Flavio Toffalini, Clémentine Maurice, Lionel Seinturier, arXiv, 2021.

[14] On Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs", Shang-Wei Lin, Jun Sun, Yang Liu, Jin Song Dong, arXiv, 2017. .

[15] TriggerZoo: A Dataset of Android Applications Automatically Infected with Logic Bombs, Jordan Samhi, Tegawendé F. Bissyandé, Jacques Klein, arXiv, 2022. https://arxiv.org/pdf/2203.04448v1.

[16] Malware Classification using Deep Neural Networks: Performance Evaluation and Applications in Edge Devices, Akhil M R, Adithya Krishna V Sharma, Harivardhan Swamy, Pavan A, Ashray Shetty, Anirudh B Sathyanarayana, arXiv, 2023 .

[17] Artificial Intelligence-Based Malware Detection, Analysis, and Mitigation,Ahmed Bouridane, Saddaf Rubab, Ibrahim Moussa Marou, Symmetry 2023, 15(3), 677, MDPI,2023.

[18] Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17). 2017. 1145-1153. doi: 10.1145/3097983.3098158.

[19] BlueKeep: A Journey from DoS to RCE (CVE-2019-0708), Exploit-DB Team, 2019,https://www.exploit-db.com/ .

[20] Analysis of CVE-2021-26897 DNS Server RCE, Ricardo Narvaja, 2021. https://www.coresecurity.com/core-labs/articles/analysis-cve-2021-26897-dns-server-rce .

[21] Integration of Static and Dynamic Analysis for Malware Family Classification with Composite Neural Network", Guolin Ke, Qiwei Ye, Taifeng Wang, Qi Meng, Weidong Ma, Tie-Yan Liu,arxiv,2019, URL: arxiv.org/abs/1912.11249 .

[22] Malware Classification using Deep Neural Networks: Performance Evaluation and Applications in Edge Devices", Akhil M R, Adithya Krishna V Sharma, Harivardhan Swamy, Pavan A, Ashray Shetty, Anirudh B Sathyanarayana,arxiv,2023, URL: arxiv.org/abs/2310.06841

[23] Towards Inspecting and Eliminating Trojan Backdoors in Deep Neural Networks,W Guo, L Wang, Y Xu, X Xing, M Du, D Song, Proceedings of the 22th IEEE International Conference on Data Mining. (ICDM'20), 2020