Article

Not peer-reviewed version

A Novel Static Analysis Approach Using System Calls for Linux IoT Malware Detection

Jayanthi Ramamoorthy * , Khushi Gupta , Ram C. Kafle , Narasimha K. Shashidhar , Cihan Varol

Posted Date: 3 July 2024

doi: 10.20944/preprints202407.0268.v1

Keywords: ELF static analysis; Linux system calls; machine learning; malware detection

Article

A Novel Static Analysis Approach Using System Calls for Linux IoT Malware Detection

Jayanthi Ramamoorthy *,† , Khushi Gupta †, Ram C Kafle †, Narasimha K Shashidhar and Cihan Varol

Department of Computer Science, Department of Mathematics and Statistics, Sam Houston State University, Huntsville, TX 77340, USA

* Correspondence: jxr153@shsu.edu; J.R.

† These authors contributed equally to this work.

Abstract: The proliferation of Internet of Things (IoT) devices on Linux platforms has heightened concerns regarding vulnerability to malware attacks. This paper introduces a novel approach to investigating the behavior of Linux IoT malware by examining syscalls and library syscall wrappers extracted through static analysis of binaries, as opposed to the conventional method of using dynamic analysis for syscall extraction. We rank and categorize Linux system calls based on their security significance, focusing on understanding malware intent without execution.Feature analysis of the assigned syscall categories and risk ranking is conducted with statistical tests to validate their effectiveness and reliability in differentiating between malware and benign binaries. Our findings demonstrate that potential threats can be reliably identified with an F1 score of 96.86%, solely by analyzing syscalls and library syscall wrappers. This method can augment traditional static analysis, providing an effective preemptive measure to enhance Linux malware analysis. This research highlights the importance of static analysis in strengthening IoT systems against emerging malware threats.

Keywords: ELF static analysis; Linux system calls; machine learning; malware detection

---

1. Introduction

Internet of Things (IoT) are reshaping cyber-physical systems with unprecedented impact. By the end of this decade, an estimated 25.44 billion devices will be interconnected, predominantly as IoT devices, comprising 75% of the total device count [1].Recent reports, including the 2023 Zscaler ThreatLabz Enterprise IoT and OT Threat Report, highlight a staggering 400% surge in attacks targeting IoT devices [2]. The rapid integration of IoT technologies across various industries signals a continued escalation in such security incidents impacting various domains such as transportation, healthcare, and energy management. However, IoT devices lack security features and are inherently complex in terms of hardware and software design, making them vulnerable to cyber-attacks [3].

Linux systems are the backbone of numerous IoT devices [4]. This ubiquity has led to a noticeable uptick in malware designed to exploit Linux-based environments, as evidenced by a 50% increase in new Linux malware reported by AV-ATLAS in just one year [5].

The focus of Malware analysis is to examine malicious binaries (malware) to discern their behavior, purpose, and impact using a range of techniques and tools. This is typically done through Static Analysis, Dynamic Analysis or a hybrid of both these approaches. The extent of analysis is only limited by the objective - to quickly detect the malware or understand its functionality and behavior patterns such as network traffic,interactions with the system, evasion techniques and other features.

In this paper, we focus solely on static analysis to extract system calls (syscalls) and library syscall wrappers made by binaries to the Linux Kernel for ARM architecture-based Linux files, commonly known as ELF (Executable and Linkable Format) binaries. We analyze 1,117 ARM-based Linux IoT malware samples and 1,214 benign or non-malicious binaries. The syscall dataset created is evaluated using common machine learning classification models, such as Logistic Regression, Neural Networks, and Random Forest, to classify these binaries as malicious or benign.

While there are numerous research works where syscalls are used to analyze malware through dynamic analysis, to the best of our knowledge, there are no known research works that extract syscalls and library syscall wrappers from static analysis. Static analysis has the limitation of not capturing all the system calls made by a binary, as it cannot detect indirect system calls. In this study, we evaluate whether the limited syscalls and library syscall wrappers that can be extracted from the disassembly of the binary are still adequate for malware detection.

The main contributions of this paper are as follows:

- **Systematic Linux syscall Categorization and Security Risk Ranking:** We categorize and assign risk rankings to all extracted calls based on their potential security risks. The validity of these categories and rankings is confirmed through statistical analysis in this study.
- **Static Analysis Dataset:** We reverse-engineered 2,331 ARM architecture-based ELF binaries to extract syscalls and library syscall wrappers using static analysis and correlated with syscall categories and security risk ranking to create a comprehensive dataset, which includes the syscalls, category and security risk ranking for each syscall, along with added statistical features for each binary.
- **Malware Detection:** To demonstrate the reliability of using static analysis for malware detection based on system calls extracted from the disassembled binaries, we evaluate the dataset created with standard ML classification models such as Logistic Regression, Random Forest classifier, Support Vector Classification (SVC), and Multi-Layer Perceptron (MLP) Neural Network.

Although we focus on ARM binaries, our syscall extraction approach is architecture-agnostic due to the use of the popular reverse engineering tool radare2. Radare2 employs the ESIL (Evaluable Strings Intermediate Language) framework, which abstracts the underlying architecture details, allowing for consistent and accurate syscall extraction across different hardware platforms [6].

Binary lifting in static analysis has been successfully employed for malware variant classification by Ramamoorthy et al [7], where the Intermediate Representation (IR) of opcode sequences was used to create a dataset. In their study, a Random Forest Classification model achieved an F1 score of 97%.

The organization of this paper is as follows: Section 2 provides the background for the study, followed by Section 3, which describes relevant research work in the field of IoT malware detection. The methodology is explained in Section 4, including statistical analysis and machine learning models used for evaluation. Section 5 presents the results obtained using the proposed methods and includes an analysis of these results. Section 6 summarizes the conclusions, followed by Section 7, which discusses the limitations and future scope.

2. Preliminaries

In this section we discuss and provide an overview of malware static analysis and system and library syscall wrappers.

2.1. System and Library Calls

In ELF ARM binaries, both library and system calls re crucial and play an important role in program functionality. They enable binaries to interact with external resources, perform operations and access system-level functionalities.

- **Syscalls:** In Linux, a system call (syscall) is the primary interface through which user-space programs request services and access functionalities from the operating system kernel. It acts as a bridge between the user space, where applications run, and the kernel space, where core system functions reside. System calls enable programs to perform privileged operations and access restricted system resources, such as hardware interaction, process management, file I/O operations, network communication, and memory management as shown in Figure 1. Some system calls are architecture-specific, and their implementation and availability can vary between different hardware platforms (e.g., x86, ARM, MIPS). These differences are due to the unique characteristics and requirements of each architecture, necessitating specific handling within the kernel to optimize performance and compatibility [8].

- **Library syscall wrappers:** When using programming languages like C++ or Java, developers often use pre-built functions from libraries such as the GNU C Library *glibc*, which contains routines for file management, memory allocation, and computational tasks. When a program calls one of these library functions, it may require system-level functionalities that reside in kernel space, like hardware interaction or process management. Consequently, library functions often make system calls and provide wrapper functions to syscalls in order to perform these tasks as shown in Figure 1.
- **Virtual syscalls:** Many Linux distributions also provide optimization of certain syscalls called virtual syscalls. Virtual syscalls, or vDSO (virtual Dynamic Shared Object) calls, are a set of performance-optimized routines provided by the Linux kernel that user-space applications can use to execute certain system calls more efficiently. These virtual syscalls are mapped directly into the process address space, allowing some system call functionalities to be executed without the overhead of a traditional syscall. vDSO calls are generally captured through dynamic analysis, as they involve runtime components and optimizations. This study does not capture Virtual syscalls.

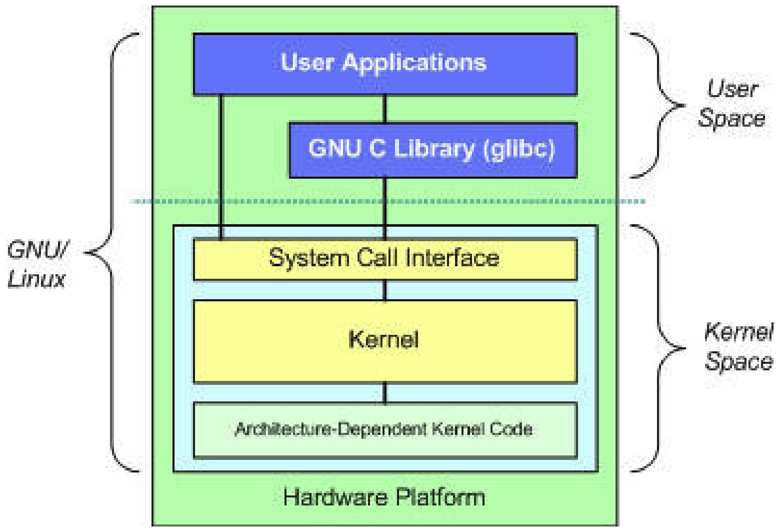


Figure 1. Architecture of the GNU/Linux operating system [9].

2.2. *Linux Malware Static Analysis*

Static malware analysis examines malicious software without execution, focusing on the code and structure of the malware to identify malicious patterns, behaviors, and potential threats. This method analyzes API calls, function calls, and data structures to understand malware logic and potential behaviors, such as file modifications, network communications, and privilege escalation attempts.

Static analysis, although powerful, has limitations - it cannot detect all system calls made during runtime, which are essential for fully understanding interactions with the operating system. Dynamic analysis complements this by capturing all interactions and changes, providing a comprehensive view of malware capabilities. The approach we propose does not seek to replace dynamic analysis; rather, it offers crucial insights into malware behavior without the need for execution.

In this paper, we introduce a novel approach that studies call patterns by extracting syscalls and library syscall wrappers from binaries through static analysis, forming the basis of the dataset used in our research.

3. Literature Review

In the past decade, malware has continued to pose a significant threat to global infrastructure. As the majority of the infrastructure systems heavily rely on Internet of Things (IoT), the importance of Linux malware analysis has become crucial, particularly because Linux is the preferred operating system for major IoT devices and servers. In this section, we present some of the previous literature on the usage syscalls to classify malware.

Asmitha et al [10,11] propose a novel non-parametric statistical approach using machine learning techniques for identifying previously unknown malicious Executable Linkable Files (ELF). They used a dataset of 226 linux malware samples and 442 benign samples from which syscalls (features) were dynamically extracted. Their proposed method prioritizes features through non-parametric statistical techniques such as the Kruskal-Wallis ranking test (KW) and Deviation From Poisson (DFP). Three learning algorithms (J48, Adaboost, and Random Forest) were used to create a prediction model using a minimal feature set extracted from the system calls. This approach yielded an optimal feature vector, resulting in an overall classification accuracy of 97.30% in identifying unknown malicious files.

Additionally, Phu et al [12] introduce a framework for analyzing MIPS ELF files based on syscall behaviors. They compiled a database of 3,773 MIPS ELF malware samples from Detux, IoTPOT, and VirusShare and execute them in a QEMU-based sandbox called F-sandbox, derived from Firmadyne and Detux. For their final feature set the authors take the 30 most common syscalls and apply the n-gram technique to construct the feature vectors. The authors then employ machine learning classifiers such as Support Vector Machines (SVM), Random Forest, and Naive Bayes to analyze the syscalls. Their experiments demonstrate that Random Forest achieves the high accuracy in classifying MIPS malware, with an accuracy of 97.44%.

Taking it a level further, Tahir and Qadir [13] propose a detection scheme for IoT malware using cross-architectural (MIPS, ARM and x86) analysis based on system calls. They had 69 system calls dynamically extracted from 1,048 samples of both IoT malware and benign binaries. They then trained five popular machine learning models: Support Vector Machine, Random Forest, Logistic Regression, Bagging, and Multi-layer Perceptron to detect the malware. The results showed that Random Forest achieved the highest detection accuracy at 99.04%. Additionally, their findings also suggested that feature sets based on system calls hold significant potential for detecting multi-architectural IoT malware.

Shobana and Poonkuzhali [14] carried out the same using deep leaning techniques. In their approach, the authors identified malware by analyzing its behavior through the sequence of system calls it generates during execution. The system calls made by IoT malware are captured using the Strace tool on Ubuntu. These recorded malicious system calls undergo preprocessing using n-gram techniques to extract necessary features. The resulting system calls are then categorized into two classes—normal and malicious sequences using a Recurrent Neural Network (RNN). The results reveal that using this technique produced 98% classification accuracy.

Abderrahmane et al [15] present a solution that will classify whether an Android application is malicious or not. The application is installed and executed simulating its usage. After execution, system calls generated by the Linux kernel are collected, processed, and fed into the neural network model that will be used to classify the application. The authors used a convolutional neural network for this research and their findings reveal that their solution could detect malicious applications with an accuracy of 93.29%.

Lastly, Ramamoorthy et al [7] focus on using binary lifting methods to perform static analysis and extract Intermediate Representation (IR) opcode sequences for analysis. The study introduces a series of statistical entropy-based characteristics derived from these opcode sequences. By concentrating solely on function metadata and opcode entropy, their approach is versatile across different architectures. It effectively identifies malware and accurately categorizes its variations, achieving an impressive F1 score of 97% .

To the best of our knowledge, existing research predominantly focuses on syscall extraction through dynamic analysis, using tools like strace to generate extensive syscall logs as shown in Table 1. This method requires executing malware within a controlled environment, which poses significant challenges in Linux due to the diversity of architectures and runtime environments. In contrast, our study uses a static analysis approach, extracting syscalls and library syscall wrappers directly from disassembled code. Although this method may not capture indirect syscalls to the kernel, we demonstrate that it remains an effective strategy for identifying malicious behavior.

Table 1. Comparative study of research works that uses syscalls for malware detection.

Research	Statistical Analysis	Architecture	Features Used	Dynamic/Static	Models	Accuracy	Comparison
Asmitha Vinod [10]	non-parametric statistical methods like Kruskal-Wallis ranking test (KW), Deviation From Poisson (DFP)	-	syscalls	Dynamic	J48, Adaboost, Random forest	97.30%	Dynamic analysis*
Asmitha Vinod [11]	-	-	syscalls	Dynamic	Naïve bayes, J48, Adaboost, RF, IBK-5	97%	Dynamic analysis*
Phu et al [12]	n-gram, chi square	MIPS	syscalls	Dynamic	RF, NB, SVM	97%	Dynamic analysis*, MIPS architecture-specific
Tahir Qadir [13]	-	x86, MIPS, ARM	syscalls	Dynamic	SVM, LR, RF, MLP, Bagging	Bagging, RF (99%)	Dynamic analysis*
Shobana Poonkuzhali [14]	-	-	syscalls	Dynamic	RNN	98.7%	Dynamic analysis*
Abderrahmane et al [15]	-	ARM	log files, syscalls	Dynamic	CNN	93.3%	Dynamic analysis*, ARM architecture-specific
Our research	chi squared test, Wilcoxon test	Multi architecture	syscalls, library syscall wrappers	Static	Linear Regression, Random Forest, SVC, Neural network	96.86%	Static analysis requiring no execution or environment setup. Multi-architecture support. Can be combined with other static features for a higher accuracy. We conduct statistical analysis to validate syscall ranking and category assignment. These scores can be adjusted for various architectures.

'*' These works use Dynamic analysis which require a compatible environment to be setup and configured for the malware to execute. Moreover dynamic analysis tends to generate a high volume of syscalls within a short period, as the kernel performs various internal operations during its execution. This rapid generation of syscalls can lead to significant noise, with numerous irrelevant and redundant syscall that can obscure meaningful patterns.

4. Methodology

In this section, we provide a detailed explanation of the workflow that was utilized throughout our research. This workflow is structured into several distinct steps, each of which is outlined in this section and visualized in Figure 2 below.

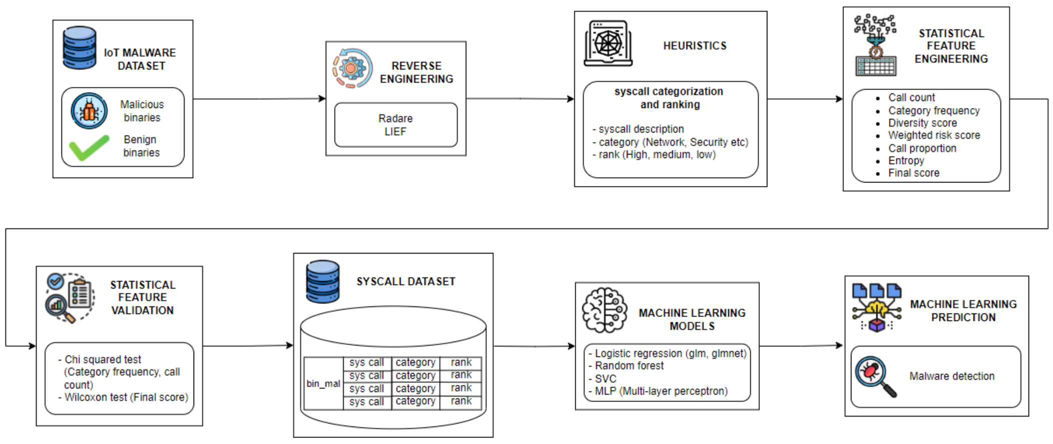

IoT MALWARE DATASET
Malicious binaries
Benign binaries
REVERSE ENGINEERING
Radare
LIEF
HEURISTICS
syscall categorization and ranking
- syscall description
- category (Network, Security etc)
- rank (High, medium, low)
STATISTICAL FEATURE ENGINEERING
Call count
Category frequency
Diversity score
Weighted risk score
Call proportion
Entropy
Final score
STATISTICAL FEATURE VALIDATION
- Chi squared test (Category frequency, call count)
- Wilcoxon test (Final score)
SYSCALL DATASET
bin_mal
sys call
category
rank
MACHINE LEARNING MODELS
- Logistic regression (glm, glmnet)
- Random forest
- SVC
- MLP (Multi-layer perceptron)
MACHINE LEARNING PREDICTION
Malware detection

Figure 2. Workflow of the proposed approach.

4.1. Iot Malware Dataset

This paper concentrates exclusively on employing static analysis to identify system calls (syscalls) and library syscall wrappers utilized by binaries targeting the Linux Kernel in ARM architecture-based Linux files, which are typically referred to as ELF (Executable and Linkable Format) binaries. Our study involves the examination of 1,117 samples of ARM-based Linux IoT malware and 1,214 binaries categorized as benign or non-malicious, with 251 unique syscalls as shown in Figure 2. The raw binaries are extracted from open-source IoT malware ARM architecture binary dataset from [16] and [17]. We selected random samples of binaries containing fewer than 2000 functions to ensure that the reverse engineering process was batched to a manageable size. Subsequent stages are illustrated as shown in Figure 2.

Table 2. Counts of binary samples.

Class	Binary Count	syscall Counts
Malware	1,117	163,288
Benign	1,214	45,284

The proportion of syscalls by ranking and categories are shown in Figures 3 and 4 which are discussed Statistical feature Analysis section.

4.2. Reverse Engineering

This study explores the identification of malicious behavior in ARM-based Linux IoT devices by analyzing syscalls and library syscall wrappers extracted solely from static analysis of ELF binaries. We employ radare2, a popular reverse engineering library, to extract syscalls and library syscall wrappers from global symbols, exports, imports, and call references within the binary.

4.3. Heuristics

Additionally, each syscall is assigned a risk level—High, Medium, or Low, reflecting its potential for malicious use. To validate the effectiveness of our categorization and ranking system, we perform a chi-squared analysis [18] to compare the distribution of syscalls between benign and malware samples. This statistical validation confirms the significant differentiation in syscall usage patterns between the malware and benign subsets.

Our dataset distribution is outlined in 2. The raw binaries are reverse engineered to extract the the following attributes: `binary_id`, `hash`, `call_api`, `call_description`, `call_category`, and `call_rank`.

Unlike Windows environment, Linux binaries have built-in debugging symbols with DWARF which is an integral part of the ELF format. This rich context contributes to gathering relevant symbols and their usage patterns, which is crucial for assessing the potential maliciousness of each call. We have systematically categorized all Linux system calls into the following ten categories based on their functionality:

Table 3. Extended categorization of Linux syscalls

Syscall category	**Description**
FileSystem	Handles file management operations such as reading, writing, and permissions.
Process	Manages process lifecycle such as creation, execution, and termination.
Memory	Controls memory allocation, deallocation, change and access critical for process management.
Network	Encompasses syscalls for network communication such as socket management, send, recv.
System	General system calls for services, system configuration and management.
Metadata	Involves retrieval and manipulation of file or system metadata.
Signal	Inter-process communication, and signal-driven interruptions.
Security	Key management, encryption, and access controls.
NonblockingIO	Non-blocking operations for input/output.
Time	Measuring time, manipulation.

4.4. Statistical Feature Engineering

To enhance the predictive power of our models, we enriched our dataset with several engineered features derived from the syscall and library call data. These features aim to capture the nuances of syscall behavior that distinguish malware from benign software. The engineered features include:

- Call Count: The total number of system calls made by each binary. This feature reflects the general activity level of the binary, which can indicate suspicious behavior.
- Distinct Call Count: The number of unique syscalls made by each binary. A high variety of calls can be indicative of complex or unusual binary behavior.
- Category Frequency: For each of the ten syscall categories, we calculated the frequency of syscalls falling into each category per binary. This helps in understanding which types of operations are predominant in a binary, aiding in profiling typical and atypical behaviors.
- High-Risk Call Proportion: The proportion of syscalls that are ranked as 'High' risk relative to the total number of syscalls. This feature specifically targets the detection of syscalls more commonly associated with malicious activities.
- Entropy of Calls: We calculated the entropy of syscall distribution within each category of a binary to measure the unpredictability and randomness of syscall usage, which can be higher in malware due to evasion techniques or diverse functionalities.
- Weighted Risk Score: By assigning weights to syscalls based on their assigned risk levels (High, Medium, Low), we computed an overall risk score for each binary. This score provides a quantitative measure of the potential threat posed by the binary based on the observed syscalls.

Each syscall in a binary is assigned a weight based on its risk rank (High, Medium, Low). The weighted rank score for each syscall can be calculated as:

$$\text{weighted\_rank\_score}_i = \text{weight}_i \cdot \text{count}_i \tag{1}$$

where weight_i is the risk weight for syscall i, and count_i is the count of syscall i within the binary. The total rank score is the sum of all weighted rank scores across all syscalls in a binary:

$$\text{total\_rank\_score} = \sum_{i=1}^{n} \text{weighted\_rank\_score}_i \tag{2}$$

where n is the number of different syscalls recorded for the binary.

The diversity score, calculated using Shannon entropy, measures the unpredictability or diversity of syscall ranks in a binary. It is defined as:

$$\text{diversity\_score} = -\sum_{j=1}^{m} p_j \log_2(p_j) \tag{3}$$

where p_j is the proportion of syscalls of rank j (e.g., High, Medium, Low), and m is the number of different ranks.

The final score is a composite metric that combines the total rank score and the diversity score, providing an overall score for each binary:

$$\text{final\_score} = \text{total\_rank\_score} + \text{diversity\_score} \tag{4}$$

These features were integrated into our analytical framework to provide a robust foundation for subsequent machine learning models to classify the samples effectively. By leveraging a combination of count-based, frequency-based, and information-theoretic metrics, we aim to capture a comprehensive profile of each binary's behavior, significantly enhancing our malware detection capabilities.

4.5. Data Organization

We index the data with a `binary_id` for organizing syscalls within each binary, to facilitate aggregation and analysis at the binary level. By grouping syscalls by their respective `binary_id`, we ensure that behavioral patterns are accurately captured for the entire binary. This structured approach enhances the reliability of our analysis and improves the model's ability to effectively learn and differentiate between distinct syscall patterns. The resulting syscall dataset created has the features as outlined in Table 4 for each `binary_id`.

Table 4. Description of the IoT Malware Syscall Analysis dataset.

Dataset feature	Description
binary_id	Unique identifier for each binary
hash	Hash of the binary
Architecture	ARM - although our metholodolgy supports multiple architectures with binary lifting strategy
isMalware	Whether the binary is malicious or beningn
prc	32-bit/64-bit
Endian	The endianess of the binary (LSB or MSB)
Stripped	Binary attributes indicating whether it is stripped of symbol information
call_api	Name of the syscall
call_desc	Description of the function of the syscall
call_type	syscall or lib (library syscall wrapper)
call_cat	Category of the syscall, Refer Table 3
call_rank	Security risk level of the syscall based on its potential for malicious use (High, Medium, Low)
bin_all_call_cnt	Count of all the syscalls and lib calls in the binary
bin_dist_call_cnt	Count of the distinct syscalls and libcalls in the binary
bin_dist_cat_cnt	Count of each category of syscalls and libcalls
bin_each_api_cnt	Count of unique syscalls in the binary
bin_calls_per_cat	Number of calls per category in a binary
bin_dist_call_per_cat	Number of calls in each distinct category in a binary
call_rank_n	Number of syscalls in the binary with the current syscall rank
bin_calls_per_rank	Number of syscalls and libcalls in each rank per binary
std_dev_calls_per_cat	Standard deviation of the syscalls per category in a binary
mean_calls_per_cat	Mean of calls per category in a binary
average_rank	Average rank of the syscalls and libcalls in a binary
weighted_rank_score	Weight of the syscall, Refer Equation 1
total_rank_score	Sum of all weighted rank scores across all syscalls in a binary, Refer Equation 2
diversity_score	Unpredictability of the syscall ranks in the binary, Refer Equation 3
final_score	Total rank score added to the diversity score, Refer Equation 4
cat_concentration_index	Number of syscall categories in the binary (Unused)

4.6. Handling of Outliers

Outliers highlight extreme behaviors or unusual patterns that are characteristic of malicious activities. Therefore, we retain outliers in our dataset to capture such behavior patterns.

4.7. Statistical Feature Analysis

To validate the effectiveness of the categorized and ranked syscalls in differentiating malware from benign binaries, we employ a chi-squared test. This statistical method evaluates whether there is a significant difference in the distribution of syscall ranks between malware and benign samples.

The test procedure is as follows:

- Data Collection: We calculate the frequency of each syscall rank for both malware and benign binaries.
- Normalization: The frequencies are normalized to proportions within each group.

- chi-squared Test: We perform the chi-squared test to determine if the observed differences in the distribution of syscall ranks are statistically significant.

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i} \tag{5}$$

where O_i are the observed frequencies of risk rank, and E_i are the expected frequencies. We use the same methodology to assess the importance of syscall categories as well.

Figure 3 shows the proportion of each syscall category for malware and benign dataset along with the chi squared test results. Similarly Figure 4 shows the syscall risk rank proportion along with the chi squared test results for the malware and benign subsets.

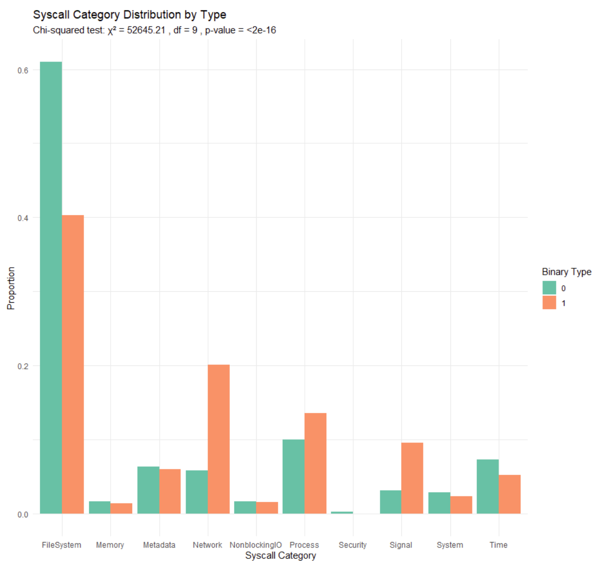


Figure 3. Proportion of syscall category in Malware and Benign subsets

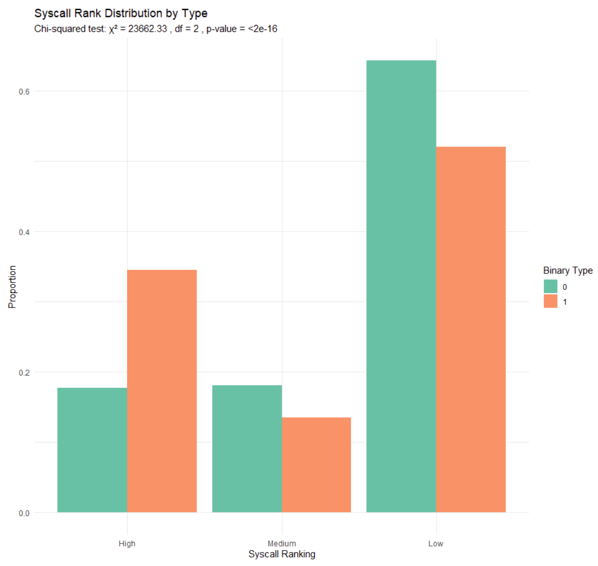


Figure 4. Proportion of security risk ranking of Syscalls in Malware and Benign subsets

The results show p-value $< 2.2 \times 10^{16}$ in both cases, implying that the rankings are not just arbitrary, and assigned rankings correspond to observable differences in behavior between malware and benign binaries. This reinforces the idea that higher risk ranked syscalls and certain categories of syscalls such as Network, Process and Signal category syscalls are more critical in the context of malware activity. Also, the High risk ranked syscalls are statistically higher in malware subset than benign binaries.

To validate the statistical significance of the differences observed between the calculated final_score based on the entropy of the syscalls and their risk ranking, we employ the Mann–Whitney (Wilcoxon) U test. This non-parametric test is suitable for skewed data for comparison from two independent groups. Our analysis reveals a negligibly low p-value $< 2 \times 10^{-16}$, confirming that the final_score computed based on the Equation 4, can reliably differentiate between malware and benign samples as shown in Figure 5.

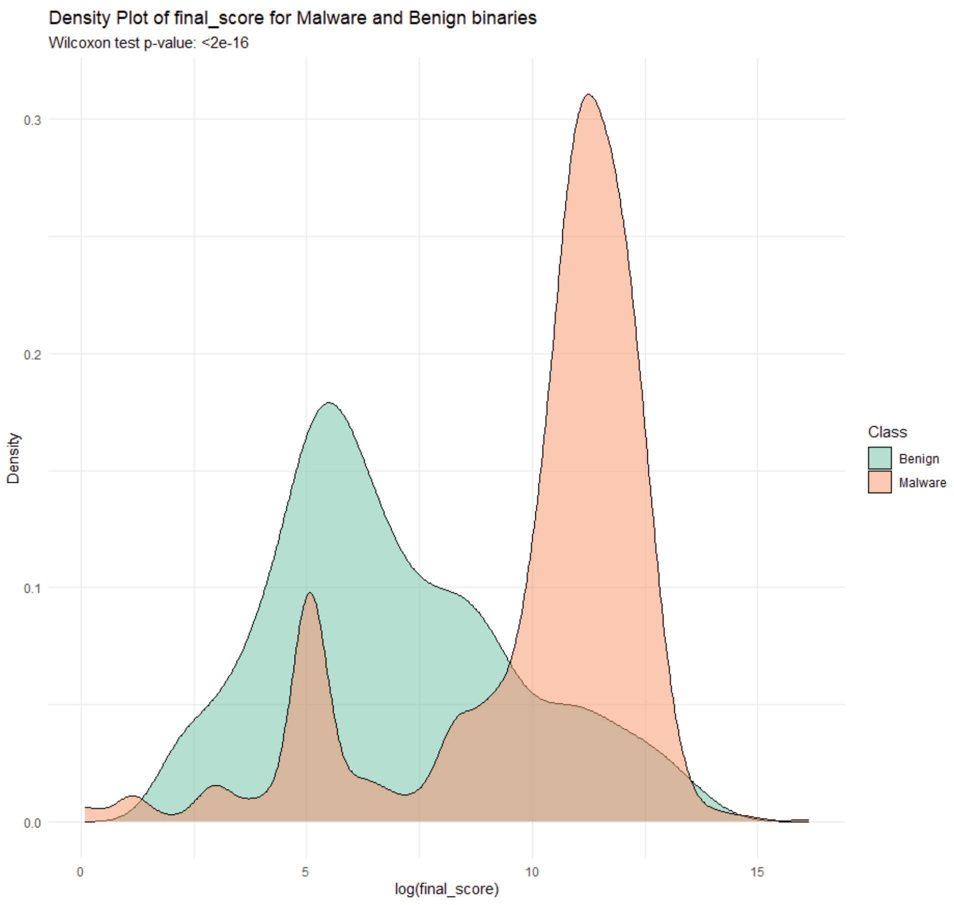


Figure 5. Differences in final-score (Equation 4) between Malware and Benign subsets.

4.8. Machine Learning Classification Based on Static Extraction of Syscalls

The primary focus of this study is to identify malware based on syscalls and library syscall wrappers extracted from static analysis as opposed to dynamic execution of the malware. Based on the extracted syscalls, we outlined the statistical features added to the dataset in previous sections, which are based on syscall categorization and security risk ranking of the syscalls used by the binary.

In this section we discuss the feasibility of Machine Learning (ML) classification models on the created dataset. We have evaluated the approach and dataset with commonly used ML classification models: Logistic Regression, Random Forest, LinearSVC, and Multi-Layer Perceptron (MLP) Neural Network.

- Logistic Regression: To evaluate a simple yet effective baseline for binary classification.
- Random Forest: Random Forest classifier is robust in handling outliers and anomalies typical of malware, and suitable for complex interactions between features.
- LinearSVC: LinearSVC is often effective with high-dimensional data, and used for interpretability.

- Multi-Layer Perceptron (MLP) Neural Network: MLP is effective is capturing complex patterns and interactions in the data through layered architecture.

As we will see in the Results section, the strong performance of all the models suggest that our approach and dataset are well-suited for malware detection.

As detailed in the Data Organization section, to preserve the context of syscalls and library syscall wrappers within each binary, we train our models using grouped data. Syscall categories and risk ranking are One-Hot encoded to handle categorical data effectively. We then compute the Variation Inflation Factor (VIF) to identify and mitigate multicollinearity in our feature set.

The Generalized linear model (GLM) using 'glmnet' statistical model package is employed with Lasso regression to train the glm model. Regularization is used to minimize overfitting and improve generalization.

The dataset is split with 80:20 ratio, ensuring a balanced representation through shuffling of batches grouped by binary_id to preserve the frequency and group of syscalls.

4.9. HyperParameters

Most of the standard default values from the 'scikit-learn' library were retained and minimal tuning was required for the ML models. In this section, ee discuss the vectorization strategy and some of the hyperparameter settings of the individual models.

4.9.1. Vectorization

For text feature 'call_api' (syscall name) we use TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer, with a maximum of 500 distinct tokens (unique Linux system calls) to extract, and an n-gram range of (1, 1). For numeric features, the mean value is used for imputation and numeric values are standardized (normalize) by removing the mean and scaling to unit variance. Categorical features, syscall category (call_cat) and syscall risk ranking (call_rank), are OneHot encoded. The pipeline for the data preprocessing and vectorization strategy is shown in Figure 6.

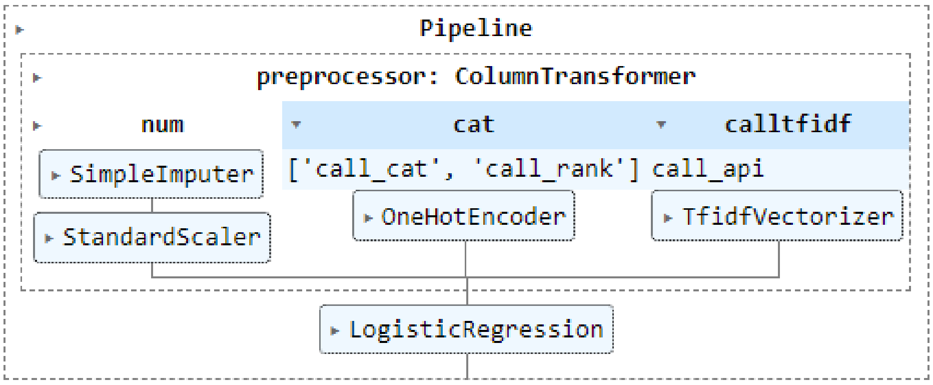


Figure 6. Preprocessing pipeline

- Linear Regression A Linear regression model is implemented using the 'scikit-learn' package [19] with the preprocessing pipeline as shown in Figure 6 The syscall imbalance between malware and benign binaries is handled with class weights assigned by the Linear regression model's balanced parameter.
- RandomForestClassifier:

  Through GridSearch, we identified the optimal number of trees as 200. The default values were retained for maximum depth of the tree, which was set to None, and Gini impurity is used for the split criteria.
- LinearSVC:

For LinearSVC, we used the default 'squared-hinge' loss function with an L2 regularization and a set the tolerance of 1×10^{-4} for the stopping criteria.

- MLP Classifier:

  For the Neural network model, we employed the rectified linear unit function (ReLU) for activation and the 'adam' solver for weight optimization, with an initial learning rate of 0.001.

Our dataset is supported by statistical features and validation which is evident from the comparable results across all ML models as discussed in the next section.

5. Results and Analysis

In this section, we compare and analyze the performance of various machine learning classification models on the syscall dataset comprising IoT malware and benign binaries. The primary objective is to evaluate the effectiveness of the ML models in distinguishing between malware and benign ELF binaries.

The performance metrics used to evaluate the models include accuracy, precision, and F1 score, defined as follows:

- Accuracy: The proportion of true results (both true positives and true negatives) among the total number of cases examined.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{6}$$

  where TP, TN, FP, and FN represent the number of true positives, true negatives, false positives, and false negatives, respectively.
- Precision: The proportion of true positive results in all positive predictions.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{7}$$

- F1 Score: The harmonic mean of precision and recall.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{8}$$

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{9}$$

The performance metrics of the ML models are shown in 5. Random Forest classifier has the best performance with a F1 score of of about 97%. The optimal threshold for the final prediction was found to be 0.72, and the maximum F1 score 96.86%. The Random Forest Classifier ROC Curve is shown in the Figure 11.

Table 5. Performance metrics of machine learning models.

Model	Accuracy	Precision	F1 Score
Random Forest	93.34%	94.71%	96.86%
Logistic Regression	92.34%	96.0%	95.06%
SVC	92.48%	96.16%	95.14%
MLP NN	92.07%	93.34%	95.03%

5.1. Discussion

As observed from the results in 5, batching syscalls for a single binary in the course of training likely plays a crucial role in all ML model's ability to detect malware binaries accurately since the the syscall frequency and order are maintained. Random Forest classifier performed the best, with a 96.8% F1 score likely due to the model's ability to handle outlier data which is characteristic of malware binaries. For example, there could be rarely used syscalls, or a direct syscall instead of library wrappers in a malware which are anomalous behaviors.

The confusion matrices for the machine learning classifiers provide detailed insights into the performance of each model in distinguishing between malware and benign binaries. The confusion matrix is a critical in understanding the classification performance, as it breaks down the results for the True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).

For example, As seen in Figure 10, Random Forest classifier was able to accurately classify 158,223 malware syscall binaries accurately with only 5,065 misclassification. Although malware binaries are characterized by extreme outliers, the Random Forest model demonstrates higher degree of accuracy and precision. The LinearSVC classifier, Logistic regression and MLP Neural Network all show similar number of True Positives as shown in Figures 7, 8 and 9 respectively.

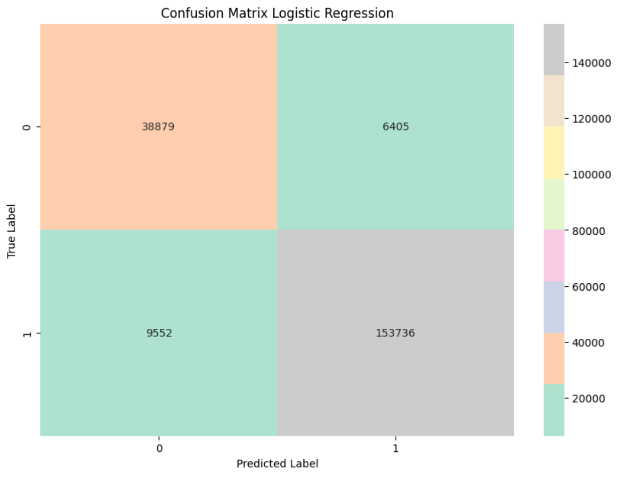


Figure 7. Logistic Regression classifier confusion matrix

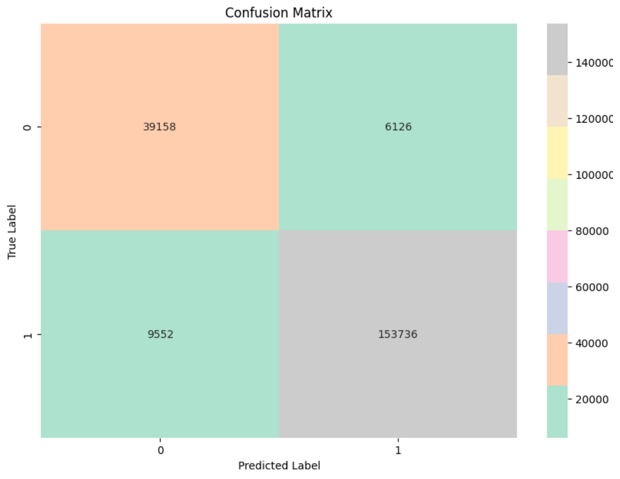


Figure 8. Linear SVC classifier confusion matrix

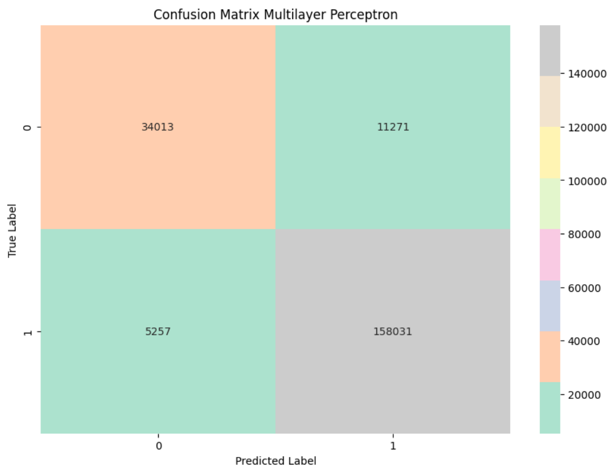


Figure 9. MLP Neural Network confusion matrix

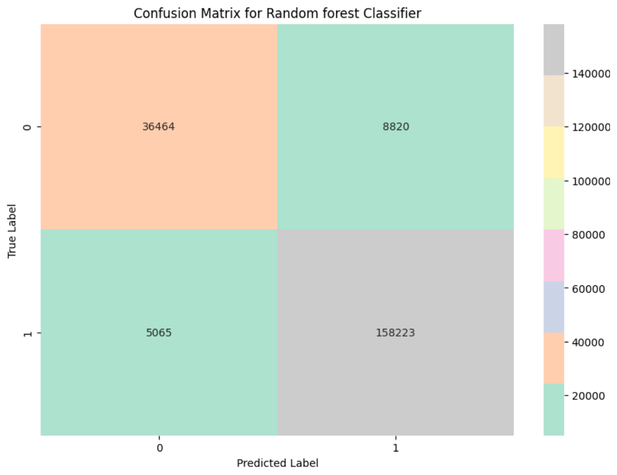
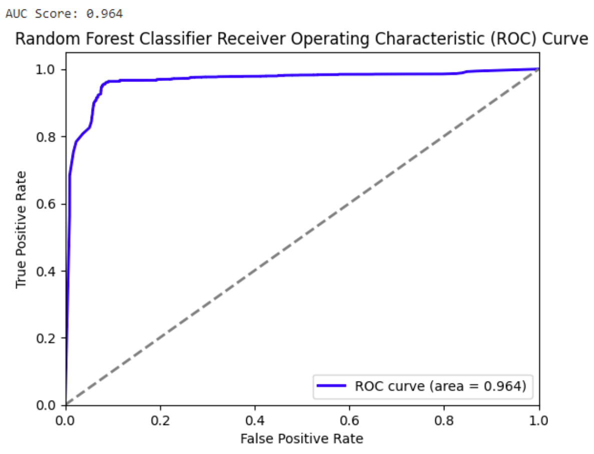


Figure 10. Random Forest classifier confusion matrix

The calculated AUC (Area Under the Curve) score for the ROC (Receiver Operating Characteristic) curve of the Random Forest classifier is 0.964, which demonstrates robust performance under various threshold as shown in Figure 11. The optimal threshold was found to be 0.72, with a F1 score of 96.8%.



```
AUC Score: 0.964
```

Figure 11. Random Forest classifier Receiver Operating Characteristic (ROC) Curve with AUC Score

The results suggest that most ML classifier models are able to accurately identify malware behavior based on static syscall analysis, since the dataset includes statistical score for the binaries based on syscall ranking and category.

6. Conclusion

This study successfully demonstrates the viability of static analysis for detecting malware in Linux IoT systems by analyzing call patterns (syscalls and library syscall wrappers) extracted from ELF binaries. By ranking and categorizing calls based on their security implications, we have established a methodological framework that distinguishes between benign and malicious binaries effectively. The application of a chi-squared test further validates the significant differences in call patterns between these groups, reinforcing the reliability of our categorization and ranking system.

We demonstrate the efficacy analyzing of call pattern usage from static analysis by using Machine learning models such as glmnet with lasso regression, Random Forest classifier, SVC and MLP neural network. The models were able to perform malware detection with a high level of accuracy, as indicated by an Random Forest classifier F1 score of 96.86%. This is particularly significant given that this study solely uses system calls extracted using static analysis without any other significant contributing features that are important in static analysis such as sections, functions, strings and file header information. By bypassing the need for binary execution, our approach mitigates the risks associated with dynamic analysis and offers an effective alternative for malware detection.

7. Future Works

While this study demonstrates the viability of static analysis for detecting malware in Linux IoT systems with a high accuracy rate, it primarily focuses on syscalls and library syscall wrappers. However, this narrow focus may not capture all of the malware behavior patterns. Future research could consider incorporating additional static features, such as section headers, functions, and strings, for a comprehensive detection strategy.

Additionally, static analysis does not capture dynamically generated syscalls, potentially leading to incomplete data. The limitations of dynamic analysis is already discussed in this study. Augmenting static analysis with dynamic analysis in future studies could provide a more comprehensive understanding of malware behavior and improve detection rates.

Building on the findings of this study, future research will aim to extend the scope of the static analysis framework. The study could be also be extended to multiple architectures for evaluation, broadening its applicability in the field of IoT malware detection.

Author Contributions: Conceptualization, J.R., R.K., K.G, and N.S.; methodology, J.R.; software, J.R.; validation, J.R., R.K., K.G, N.S., and C.V.; formal analysis, J.R.; investigation, J.R. and R.K.; resources, N.S., and C.V.; data curation, J.R.; writing—original draft preparation, J.R., and K.G.; writing—review and editing, K.G., and J.R.; visualization, J.R., K.G, R.K.; supervision, R.K and N.S.; project administration, C.V.; funding acquisition, C.V.

Funding: This research received no external funding

Data Availability Statement: The syscall dataset created in the study is openly available at:
https://github.com/jrmoorthy/Linux-Malware-Analysis

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

References

1. Howarth, J. 80+ Amazing IoT Statistics (2024-2030) — explodingtopics.com. https://explodingtopics.com/blog/iot-stats. [Accessed 09-05-2024].
2. Zscaler ThreatLabz Finds a 400Year-over-Year. https://www.zscaler.com/press/zscaler-threatlabz-finds-400-increase-iot-and-ot-malware-attacks-year-over-year-underscoring. [Accessed 09-05-2024].
3. Ngo, Q.D.; Nguyen, H.T.; Le, V.H.; Nguyen, D.H. A survey of IoT malware and detection methods based on static features. *ICT express* **2020**, *6*, 280–286.
4. Antony, A.; Sarika, S. A review on IoT operating systems. *Int. J. Comput. Appl* **2020**, *176*, 33–40.
5. AV-ATLAS Malware Portal. https://portal.av-atlas.org/malware, 2023. [Online; accessed 10-May-2023].

6. Pancake.; others. The Official Radare2 Book: ESIL, 2024.
7. Ramamoorthy, J.; Gupta, K.; Shashidhar, N.K.; Varol, C. Linux IoT Malware Variant Classification Using Binary Lifting and Opcode Entropy. *Electronics* **2024**, *13*, 2381.
8. Kerrisk, M.; others. Linux Programmer's Manual: syscalls, 2024. Online; accessed 18-June-2024.
9. Jones, M. IBM Developer — developer.ibm.com. https://developer.ibm.com/articles/l-linux-kernel/. [Accessed 09-05-2024].
10. Asmitha, K.; Vinod, P. Linux malware detection using non-parametric statistical methods. 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE, 2014, pp. 356–361.
11. Asmitha, K.; Vinod, P. A machine learning approach for linux malware detection. 2014 international conference on issues and challenges in intelligent computing techniques (ICICT). IEEE, 2014, pp. 825–830.
12. Phu, T.N.; Dang, K.H.; Quoc, D.N.; Dai, N.T.; Binh, N.N. A novel framework to classify malware in mips architecture-based iot devices. *Security and Communication Networks* **2019**, *2019*, 1–13.
13. Tahir, I.; Qadir, S. Machine Learning-based Detection of IoT Malware using System Call Data **2022**.
14. Shobana, M.; Poonkuzhali, S. A novel approach to detect IoT malware by system calls using Deep learning techniques. 2020 International Conference on Innovative Trends in Information Technology (ICITIIT). IEEE, 2020, pp. 1–5.
15. Abderrahmane, A.; Adnane, G.; Yacine, C.; Khireddine, G. Android malware detection based on system calls analysis and CNN classification. 2019 IEEE wireless communications and networking conference workshop (WCNCW). IEEE, 2019, pp. 1–6.
16. Olsen, S.H.; OConnor, T. Toward a Labeled Dataset of IoT Malware Features. 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 2023, pp. 924–933.
17. Refade. IoT_ARM: A Collection of IoT Malware Samples for ARM Architecture, 2024.
18. Tallarida, R.J.; Murray, R.B.; Tallarida, R.J.; Murray, R.B. Chi-square test. *Manual of pharmacologic calculations: with computer programs* **1987**, pp. 140–142.
19. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* **2011**, *12*, 2825–2830.