Learning from system traces of Android Malware

Uriel Antonio Buitrago

I. Introduction

THIS paper presents an analysis of common Machine Learning algorithms and techniques applied to the Linux/Android Kernel. The main objective is to evaluate the effectiveness of supervised learning techniques when applied to system traces of Android malware. Within the context of this paper, *system traces* refer to captured system calls and binder processes made by individual APKs (Android Package Kit file format). The traces were sourced from the publicly available CICMalDroid2020 dataset [1] [2]. This dataset contains 17,341 Android APKs categorized in five distinct classes: Adware, Banking malware, SMS malware, Riskware, and Benign. The motivation for this work is to establish a baseline supervised learning model capable of classifying common Android malware families based on low-level device activity. A Support Vector Machine (SVM), Random Forest Classifiers (RFC), Multinomial Logistic Regression (LR), and Ensemble methods are evaluated by their performance in each malware class.

The paper starts by explaining necessary Android components, vulnerabilities, and malware. A brief literature review then presents current bodies of work which have contributed to the field. Specifically, an overview of how machine learning detects malicious APKs. The paper proceeds to describe the methodology, beginning with the data collection process where numerous Android APKs, both benign and malicious, are represented by their feature vectors. The data is then processed for feature selection, and normalized to ensure that the range of the dataset features does not skew the results of the machine learning algorithms. In the training model section, performance is optimized through hyper-parameter tuning, utilizing techniques such as grid search and cross-validation to find the optimal settings. For evaluation, the paper presents a comparative analysis of each model's effectiveness based on metrics such as accuracy, precision, recall, and F1 score. The conclusion summarizes the findings, emphasizing the significant role of machine learning in enhancing Android security. It reflects on the impact of feature selection and the importance of a comprehensive dataset that captures the diverse nature of Android malware. The paper concludes with suggestions for future work, including the integration of more dynamic features, and the exploration of unsupervised learning models for anomaly detection.

A. Background

Android, being the dominant mobile operating system with over 3 billion users worldwide, has attracted many bad actors who seek to abuse mobile marketplaces such as Google Play. Studies have shown these marketplaces host presumably legitimate apps which actually contain embedded malware [3]. Additionally, Android allows for side-loading code. This means a user can install an application APK from a source that is not a centralized marketplace like Google Play. This greatly exacerbates the threat landscape an attack vectors. In combination with the increasing vulnerabilities in the Android Kernel [4], kernel-targeted attacks initiated through system calls can be perceived as inconspicuous application activity.

System calls (syscalls) are the mechanism through which user-space applications request services from the operating system's kernel, such as file operations, process management, or network communication. When a syscall is made, the application provides a specific request number and parameters; the system then switches to kernel mode to safely execute the request, returning to user mode upon completion. Unfortunately, this interface may provide a useful attack vector for compromising system security.

The Android Binder is a crucial IPC (Inter-Process Communication) mechanism that facilitates robust interaction between processes. It operates through the Binder kernel driver, which creates a device file /dev/binder to bridge communication gaps between processes. The Binder driver ensures that data can be securely copied from one process's memory space to another, safeguarding the integrity and isolation of processes. User space programs interact with the Binder driver using system calls such as open, mmap, and ioctl, which allows them to utilize the services provided by the Binder infrastructure.

1) Literature Review: Previous works [5] [6] have established that syscall execution traces are valuable context for modeling system behaviour to detect malware early. Zhang et al. presents a novel approach to identifying Android malware by analyzing system call

traces using a Multi-layer Perceptron (MLP) model [7]. The paper highlights the development of an Android malware detection system (AMDS) that demonstrates high accuracy in early malware detection using traces of the first 3000 system calls, achieving an average accuracy of 99.34%. They also implemented a Client-Server based app that used the MLP model to classify unseen Android apps as benign or malware. Hou et al. also deployed a Machine Learning grounded application, Deep4MalDroid, for early detection of malware [8]. It is a Deep Learning framework for Android Malware Detection Based on Linux Kernel system call graphs. With promising experimental results on data sampled from Comodo Cloud Security Center, they integrated Deep4MalDroid into a commercial Android anti-malware software. CopperDroid tracks and analyzes the system calls invoked by Android malware to understand their behaviors. By automatically unmarshalling inter-process communication and remote procedure call interactions, CopperDroid reconstructs complex Android-specific behaviors, demonstrating that all behaviors can be traced back to the invocation of system calls [5]. System calls serve as the underlying mechanism through which all behaviors of Android applications, including malware, are manifested.

II. Learning Setup

A. Problem Statement

This study will leverage the CICMalDroid2020 dataset, created by CopperDroid, to determine which combination of system traces and models produce the best classifier. The dataset provides CSV files with the frequency count of low-level activity (syscalls and binder interactions) to create feature vectors for each APK. In other words, the rows represent a single APK (bening or malicious), with each column containing the frequency count for every possible system call (139 unique syscalls) and binder interaction (331 possible). The final column is the classification label (1-5) for the traced APK. Table I presents the possible classification labels with their meaning.

Label	Malware Family	Description
1	Adware	Software that automatically displays or downloads advertising material.
2	Banking Trojans	Malicious programs designed to steal your banking credentials.
3	SMS	Malware that sends or intercepts SMS messages without user permission.
4	Riskware	Legitimate software that poses potential risks if exploited.
5	Benign	Harmless applications that are not considered malware.

TABLE I: Mapping of Label IDs to Malware Families with Descriptions

B. Data Processing and Tools

The following tools and libraries were used for training and visualization: Google Colab Pro, scikit-learn, numpy, pandas, optuna, matplotlib, and seaborn. They were also essential in data pre-processing given how complex it was.

CICMalDroid2020 contains two CSV files which create different feature spaces. One file, named "feature-vectors-syscalls-frequency-5-Cat.csv" contains the frequency of 139 uniqe system calls for each APK. The other file, which concatenates "binders" to the filename, also posses the syscall frequency, in addition to the binder events. I wanted to determine which feature space of the two provided a better direction for training. To do so, I trained a random forest on both datasets, utilizing all the features, but still creating training and testing splits. Because the larger feature space (syscalls and binders) produced a better RFC model, I carried out the remainder of the study with the larger CSV file.

The original dataset contains 471 feature columns from 11,598 APK traces (rows). This is a moderately sized dataset which could present bottle-necks in training. To reduce noise and improve classification accuracy, I first normalized the data and then performed feature selection. L2 normalization was applied to scale the value of features between [0,1], creating consistency. In feature selection, my aim was to select the features which are highly dependent on the response. I used Chi-Square scores for the feature selection. A high Chi-Square value indicates that the feature is more dependent on the response, and so we select it for model training. Additionally, I applied Recursive Feature Elimination to improve the Logistic Regression model. Table II shows the features selected by their Chi-Squared score.

III. Evaluation

Overall, nine models were evaluated. Three were Random Forest Classifiers, two were Support Vector Machines, two were Decision Tree ensembles, and two were Multinomial Logistic Regression. The following sections detail how each model was trained, tuned, and evaluated.

A. Random Forest Classifiers

The initial study commenced with Random Forest Classifiers because they are adept in handling large datasets with class imbalances. I trained three variations

TABLE II: Top 30 features with highest chi-square score.

Feature	Chi2Score	P-Value	Abbreviatio
read	266.801731	0.000000	F1
prctl	189.892280	0.000000	F2
fcntl64	187.269878	0.000000	F3
sigprocmask	152.554088	0.000000	F4
gettimeofday	139.254479	0.000000	F5
sendto	131.001354	0.000000	F6
pwrite64	104.796420	0.000000	F7
clock_gettime	101.995143	0.000000	F8
recvfrom	92.355911	0.000000	F9
writev	87.910064	0.000000	F10
fstat64	84.906733	0.000000	F11
mmap2	65.179863	0.000000	F12
close	62.778122	0.000000	F13
ioctl	59.879551	0.000000	F14
__arm_nr_cacheflush	56.017766	0.000000	F15
inotify_add_watch	46.380736	0.000000	F16
epoll_wait	43.823871	0.000000	F17
fdatasync	36.579841	0.000000	F18
getLine1Number	34.542882	0.000001	F19
madvise	32.165812	0.000002	F20
nanosleep	30.683091	0.000004	F21
fchown32	29.313129	0.000007	F22
lstat64	27.374234	0.000017	F23
munmap	27.356828	0.000017	F24
futex	26.754301	0.000022	F25
mprotect	25.479219	0.000040	F26
FS_ACCESS(READ)_____	22.692992	0.000146	F27
open	22.650944	0.000149	F28
gettid	21.191203	0.000290	F29
pread64	20.551758	0.000389	F30

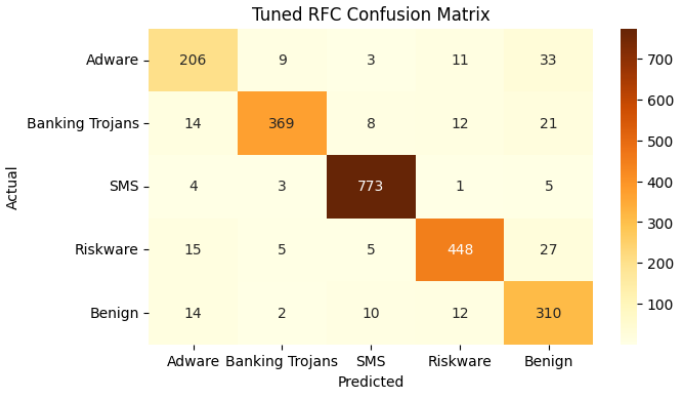

Fig. 1: RFC tuned with Top 30 features Confusion Matrix

of RFC: RFC with all features (471), RFC tuned on all features, and RFC tuned on the top 30 features. To my surprise, the best performing random forest was actually trained on the ENTIRE feature set. It achieved an accuracy of 0.94 without even tuning the hyper parameters. For consistency sake, I will focus on the results of models trained on the top 30 features from Table I. Refer to the Appendix C for all the classification results. Figure 1 details the Confusion Matrix for the RFC tuned on the top 30 features. The dataset was split into training and testing sets. A GridSearchCV was employed to fine-tune the RFC by optimizing the number of trees to 40 with 5-fold cross-validation on the training data. The best-performing model was identified based on accuracy, and its predictions were evaluated against the test set. From the confusion matrix, it's apparent that the model performs well on 'SMS' and 'Riskware' categories, with high true positive rates, but struggles with 'Adware' and 'Benign' classes, where false negatives are more prevalent. This suggests that the model may benefit from further parameter tuning or class-specific adjustments to improve its ability to distinguish between these classes. One benefit of this model is that it is quick to train and do a grid search with.

B. *Support Vector Machines*

The initial SVM model was trained using a pipeline that standardized the feature set and used a default C parameter (regularization strength). This initial model achieved an accuracy of approximately 80.95%, with varying precision and recall across different classes, indicating a need for improvement, especially for the 'Adware' class. To optimize the model's performance, hyper-parameter tuning was conducted for the regularization strength. Using the Optuna framework, a study with 100 trials was performed using 5-fold cross-validation to identify the best C value across a logarithmic scale from 1e4 to 1e-4. This process resulted in identifying an optimal C value that improved the model's accuracy to approximately 88.97%.Retraining the SVM with this optimized parameter led to improvements in all evaluation metrics. The accuracy of the model increased by about 8%, and there were notable improvements in precision and recall for all classes, especially 'Adware', which saw an increase in both precision and recall. This tuning process demonstrates the effectiveness of hyper-parameter optimization in enhancing model performance, particularly in achieving a more balanced classification across the various classes.

C. *Multinomial Logistic Regression*

The initial model was a multinomial logistic regression, trained on the top 30 features determined by

Chi-square feature selection. It achieved an accuracy of 68.02% on the test data. The classification report revealed weaknesses in precision and recall for certain classes, notably Adware (class 1) which had the lowest recall, suggesting difficulty in identifying this category correctly.

To improve the model, a two-phase tuning process was executed. The first phase used GridSearchCV to optimize the regularization strength C over a specified range. This resulted in a best C value that aimed to balance the bias-variance trade-off. In the second phase, Recursive Feature Elimination with Cross-Validation (RFECV) was applied to further refine the feature set, resulting in an optimal subset of 27 features. These features were deemed most significant in predicting the target variable. Figure 2 depicts the improvements of the mean cross-validation scores as a function of the features selected via RFE.

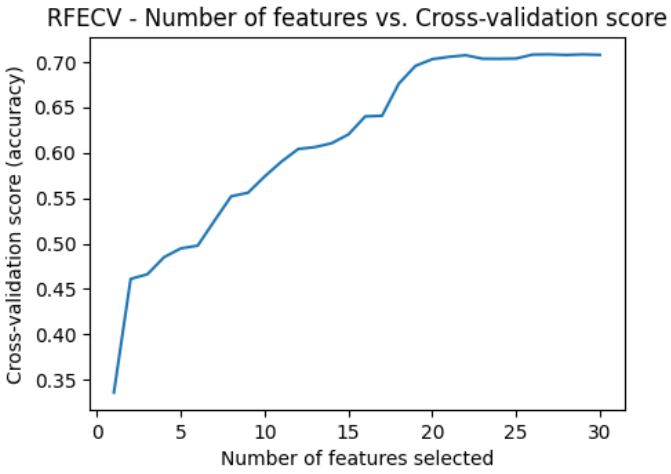

Fig. 2: Logistic Regression Tuning

After tuning, the model exhibited a slight increase in accuracy to 69.18%, with improvements in precision and recall across several classes. The RFECV process also helped in understanding feature importance, potentially offering insights into the underlying data structure and the behavior of different software classes. While there was an increase in overall accuracy, the improvements were modest, indicating that the logistic regression model has limitations in handling this particular classification problem, or that further feature engineering and model tuning might be necessary to achieve significant performance gains. However from my experience, attempting to train RFECV on the original feature space (471 columns) was very computationally expensive. After disconnecting from the runtime several times and using up most of my compute credits, I decided to train with only the top 30 features.

D. Decision Tree Ensembles

Decision trees are commonly used as base estimators for ensemble methods like bagging and boosting because of their high variance and low bias. The focus for me was to compare the performance of two ensemble methods, AdaBoost and Bagging, both utilizing a Decision Tree Classifier as their base estimator. The comparison was done using cross-validation to determine the optimal maximum depth of the decision trees, which is a key hyper-parameter that can affect model complexity and the risk of over-fitting. The AdaBoost model achieved

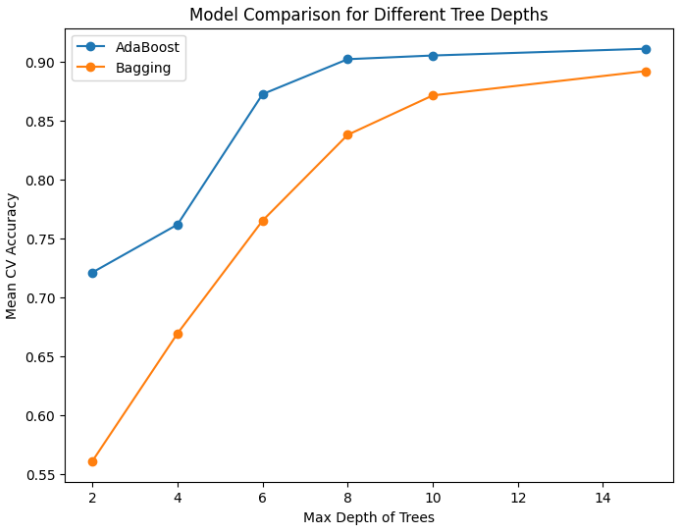

Fig. 3: Adaboost vs Bagging GridSearchCV

a higher cross-validation accuracy and test accuracy (approximately 91.55%) compared to the Bagging model (approximately 89.22%). This performance was assessed over a range of tree depths, as depicted in Figure 3, where both models generally showed improved performance with increased tree depth, up to a point. AdaBoost's accuracy plateaued beyond a depth of 10, while Bagging's improvements slowed after a depth of 6, suggesting that a larger depth provides diminishing returns or potential overfitting for the Bagging model.

The results indicate that AdaBoost, which focuses on sequentially correcting the mistakes of weak learners, may be more effective for this particular dataset when using deeper trees. On the other hand, Bagging, which reduces variance through averaging the predictions of multiple trees, shows competitive performance but does not quite match the accuracy of AdaBoost with the same base learner configuration. This comparison highlights the strengths of boosting techniques in iterative improvement and the robustness of bagging with shallower trees.

IV. Conclusions

The classification performance of various models on malware detection tasks was compared in Appendix C,

focusing on the metrics of precision, recall, F1-score, and overall accuracy. The Random Forest Classifier (RFC) with all features and tuned RFC with the top 30 features stand out with the highest accuracy of 94%. Both versions of the Support Vector Machine (SVC) and Logistic Regression (LogReg) models showed lower accuracy, with the tuned versions demonstrating improvements over the initial models. The AdaBoost ensemble method, particularly when tuned with the top 30 features, shows excellent performance across all categories, closely matching the RFC models' accuracy.

In the case of specific malware types, the AdaBoost classifier with the top 30 features shows superior performance for 'Adware' and 'Banking Trojan' categories when it comes to recall, suggesting its robustness in identifying these malware types. The Bagging ensemble method exhibits a balanced performance across different malware types with a slight trade-off in accuracy compared to AdaBoost.

Future work could explore several avenues to improve classification performance further. Experimenting with additional feature selection methods or feature transformations could yield more discriminate features and potentially enhance model performance, especially for models that did not perform as well, like the Logistic Regression. Further tuning of hyper-parameters beyond the explored C parameter for SVM and tree depth for AdaBoost and Bagging could refine models' abilities to generalize. Exploring deep learning models, which can automatically learn feature hierarchies, may provide improvements over traditional machine learning methods, especially in complex feature spaces. This has already shown success in other works like Deep4MalDroid discussed in the literature review.

REFERENCES

[1] S. Mahdavifar, A. F. Abdul Kadir, R. Fatemi, D. Alhadidi, and A. A. Ghorbani, "Dynamic android malware category classification using semi-supervised deep learning," in *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pp. 515–522, 2020.

[2] S. Mahdavifar, D. Alhadidi, and A. A. Ghorbani, "Effective and efficient hybrid android malware classification using pseudo-label stacked auto-encoder," *J. Netw. Syst. Manage.*, vol. 30, jan 2022.

[3] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, (New York, NY, USA), p. 317–326, Association for Computing Machinery, 2012.

[4] Y. Niu, L. Lei, Y. Wang, J. Chang, S. Jia, and C. Kou, "Sasak: Shrinking the attack surface for android kernel with stricter "seccomp" restrictions," in *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*, pp. 387–394, 2020.

[5] K. Tam, A. Fattori, S. Khan, and L. Cavallaro, *CopperDroid: Automatic Reconstruction of Android Malware Behaviors*, pp. 1–15. NDSS Symposium 2015, Feb. 2015.

[6] M. Zipperle, F. Gottwalt, E. Chang, and T. Dillon, "Provenance-based intrusion detection systems: A survey," *ACM Comput. Surv.*, vol. 55, dec 2022.

[7] X. Zhang, A. Mathur, L. Zhao, S. Rahmat, Q. Niyaz, A. Javaid, and X. Yang, "An early detection of android malware using system calls based machine learning model," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[8] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," in *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, pp. 104–111, 2016.

APPENDIX

A. Notebook Source

https://github.com/ubuitrago/MiLK

B. Tuning AdaBoost

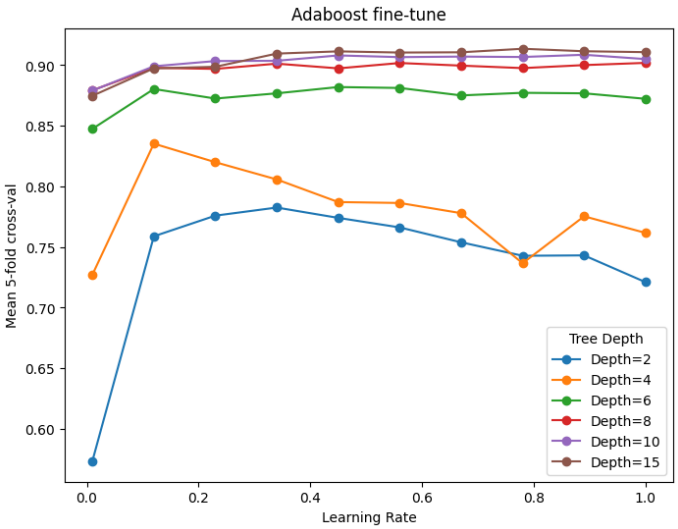

Fig. 4: Adaboost GridSearchCV

C. Combined Classification Reports

TABLE III: Classification Reports by Model and Malware

Models	Adware			Banking Trojan			SMS			Riskware			Benign			Accuracy
	Prec.	Recall	F1	Prec.	Recall	F1	Prec.	Recall	F1	Prec.	Recall	F1	Prec.	Recall	F1	
RFC_allfeatures	0.88	0.94	0.91	0.98	0.89	0.93	0.97	0.99	0.98	0.94	0.92	0.93	0.89	0.94	0.91	0.94
RFC_allfeatures_tuned	0.87	0.95	0.91	0.97	0.89	0.93	0.98	0.99	0.98	0.96	0.93	0.94	0.90	0.95	0.92	0.95
SVC_top30_tuned	0.77	0.82	0.79	0.88	0.87	0.87	0.97	0.99	0.98	0.89	0.87	0.88	0.82	0.78	0.80	0.89
LogReg_top30	0.62	0.32	0.42	0.67	0.56	0.61	0.76	0.91	0.83	0.64	0.66	0.65	0.57	0.62	0.59	0.68
LogReg_RFE_tuned	0.56	0.29	0.39	0.67	0.62	0.65	0.77	0.91	0.84	0.67	0.65	0.66	0.59	0.64	0.61	0.69
Adaboost_top30_tuned	0.85	0.78	0.81	0.95	0.87	0.91	0.98	0.99	0.98	0.91	0.91	0.91	0.80	0.92	0.86	0.92
AdaBoosttop30_tuned	0.86	0.79	0.82	0.95	0.87	0.91	0.97	0.99	0.98	0.92	0.91	0.91	0.79	0.92	0.85	0.92
Baggingtop30_tuned	0.83	0.74	0.79	0.91	0.86	0.89	0.95	0.98	0.96	0.90	0.88	0.89	0.78	0.86	0.82	0.89
RFC_top30_tuned	0.81	0.79	0.80	0.95	0.87	0.91	0.97	0.98	0.98	0.93	0.90	0.91	0.78	0.89	0.83	0.91