Interpretable Android Malware Detection Based on Dynamic Analysis

Arunab Singh, Maryam Tanha, Yashsvi Girdhar and Aaron Hunter
School of Computing and Academic Studies, British Columbia Institute of Technology, Canada

Keywords: Malware, Dynamic Analysis, Android, Security.

Abstract: Android has emerged as the dominant operating system for smart devices, which has consequently led to the proliferation of Android malware. In response to this, different analysis methods have been suggested for the identification of Android malware. In this paper, we focus on so-called dynamic analysis, in which we run applications and monitor their behaviour at run-time rather analyzing the source code and resources (which is called static analysis). One approach to dynamic analysis is to use machine learning methods to identify malware; essentially we run a large set of applications that may or may not be malware, and we learn how to tell them apart. While this approach has been successfully applied, both academic and industrial stakeholders exhibit a stronger interest in comprehending the rationale behind the classification of apps as malicious. This falls under the domain of interpretable machine learning, with a specific focus on the research field of mobile malware detection. To fill this gap, we propose an explainable ML-based dynamic analysis framework for Android malware. Our approach provides explanations for the classification results by indicating the features that are contributing the most to the detection result. The quality of explanations are assessed using stability metrics.

1 INTRODUCTION

Mobile malware poses substantial security risks, impacting both individuals and enterprises by exploiting valuable data stored on mobile devices. Android, holding a 72% market share (Statista, 2023), is a prime target due to its open nature. Cyberattacks, such as mobile banking threats (Kapersky, 2023), underscore this vulnerability despite efforts by Google and OEMs (original equipment manufacturers) (Kovacs, ByEduard, 2023).

Dynamic analysis is an important technique for Android malware detection. This approach involves running potentially malicious Android applications in a controlled environment to monitor their behavior in real-time. By observing how an app interacts with the device, network, and other applications, security analysts can uncover hidden malicious activities that may not be evident through static analysis alone. Dynamic analysis provides valuable insights into an app's runtime behavior, enabling the detection of malware that exhibits polymorphic or obfuscated characteristics.

There is a great deal of research on Android malware detection through dynamic analysis, much of which is based on using machine learning (ML) techniques (Yan and Yan, 2018). While ML models show promise in detecting malicious apps, their underlying selection processes are often complex, making it challenging to explain why an app has been classified as malicious. This lack of transparency hinders the ability to understand the key features or behaviors that triggered the classification result, which is crucial for both security analysts and app developers. Moreover, the absence of explainability makes it difficult to identify false positives and false negatives, potentially damaging the reputation of legitimate apps and allowing some malicious ones to remain undetected. Researchers are actively working on methods to enhance the interpretability of ML models, as it's not only about making accurate predictions but also about providing insights into why certain decisions are made, thereby improving the overall effectiveness and trustworthiness of ML-based systems. Explainable ML methods have not been employed extensively for dynamic analysis. Moreover, some of the existing research studies on dynamic analysis rely on inaccessible or outdated tools. A notable trend is the scarcity of open-source initiatives or comprehensive instructions, hindering result reproduction and validation.

The main contributions of this paper are summarized as follows:

- We present a clear methodology for producing a tool for dynamic malware analysis. The list of

Singh, A., Tanha, M., Girdhar, Y. and Hunter, A.
Interpretable Android Malware Detection Based on Dynamic Analysis.
DOI: 10.5220/0012415800003648

In *Proceedings of the 10th International Conference on Information Systems Security and Privacy (ICISSP 2024)*, pages 195-202
ISBN: 978-989-758-683-5; ISSN: 2184-4356

apps used, the feature extraction and selection, and developed ML models (for which we also addressed the class imbalance issues to improve the results) are open-source and publicly available to the research community on GitHub [1]. This will facilitate the reproducibility of our research.

- We employ an explanation technique for interpreting the predictions made by our ML models and evaluate its effectiveness. In this way, we can identify the features that are contributing the most to detection of an app as malware as well as evaluating the quality of explanations. To the best of our knowledge, this study is the first work attempting to provide a systematic approach to use explainable ML methods for dynamic analysis of Android malware.
- We have created datasets of traces for recent Android apps (goodware and malware), which is well-suited for dynamic analysis. Also, using our open-source scripts, researchers can generate new datasets in an easy and seamless manner.

2 BACKGROUND AND RELATED WORK

2.1 Preliminaries

There are two main approaches to Android malware detection and analysis: static analysis and dynamic analysis. The *static analysis* approach involves analyzing the code and resources of an app without executing it. One problem with static analysis is that it is generally unable to detect new malicious behaviour that happens at run-time (Yan and Yan, 2018). It is also not effective in identifying malicious deformation techniques such as java reflection and dynamic code loading (Pan et al., 2020). Hence, a comprehensive approach to malware analysis can not be based solely on static methods. *Dynamic analysis* refers to the approach where the behaviour of an application is monitored while it is executed in a controlled environment. In this manner, we can monitor things like system calls, network traffic, user interactions and inter-process communication. One advantage of this approach is that it can identify zero-day attacks that have not previously been observed. Moreover, it can identify malware that has been deliberately written to evade static analysis(Yan and Yan, 2018). Compared with static analysis, there are fewer studies on dynamic analysis and it needs further investigation by the research community. Therefore, in this paper, our focus is on *dynamic analysis* for Android malware detection.

[1] https://github.com/maryam-tanha/DynamicAndroidMalwareAnalysis

2.2 Interpretable Machine Learning for Dynamic Analysis

There are many studies that have used ML methods (including deep learning) for dynamic analysis of Android malware. However, researchers have raised concerns regarding the over-optimistic classification results produced by various ML-based malware detection techniques (Arp et al., 2022; Pendlebury et al., 2019). Furthermore, the approaches to Android malware detection mainly rely on black-box models. Consequently, security analysts frequently found themselves questioning the reliability of predictions from such highly accurate ML-based malware detection models and pondering the suitability of model selection before deployment. In response to these issues, several studies introduced methods aimed at explaining the predictions made by ML-based malware detection models. Examples of such studies for static analysis are (Fan et al., 2020; Wu et al., 2021; Liu et al., 2022b). While there are studies that attempt at providing explanations for dynamic analysis such as (De Lorenzo et al., 2020), none of the existing ones have investigated using eXplainable Artificial Intelligence techniques(XAI) (Dwivedi et al., 2023) for interpreting the results of classification for dynamic analysis.

2.3 Using System Calls for ML-Based Dynamic Analysis

In the following, we give a brief summary of the main recent studies that employed system calls as their only set of features or along with other features. We refer interested readers to (Yan and Yan, 2018; Liu et al., 2022a; Razgallah et al., 2021) for more comprehensive surveys of dynamic analysis for Android malware detection. SpyDroid (Iqbal and Zulkernine, 2018), is an Android malware dynamic analysis framework that utilizes multiple malware detectors. It supports monitoring CPU and memory usage as well as kernel system calls.Android malware detection system (AMDS) (Zhang et al., 2022) relies on system call traces, which are processed using N-gram analysis. The experimental findings illustrate that AMDS exhibits the ability to detect threats at an early stage with high accuracy. TwinDroid (Asma Razagallah, 2022) is a dataset of 1000 system call traces for Android apps. An early version of TwinDroid is

used in (Razgallah and Khoury, 2021) for Android malware detection by using n-grams as well as Term Frequency–Inverse Document Frequency (TF–IDF) weight vector of system calls and ML classification methods. MALINE (Dimjašević et al., 2016) employs system call frequency and system call dependency for indicating malicious behaviour. Another recent study, (Mahdavifar et al., 2020), detects malware families based on different categories of system calls and applying a semi-supervised deep learning method. Sequences of system calls are used in Deep4MalDroid (Hou et al., 2016) to construct a weighted directed graph, which is then given to a deep learning framework for malware detection. DroidScribe (Dash et al., 2016) provides a framework for malware family classification. It utilizes (Tam et al., 2015) as its dynamic analysis component to capture system calls information.

3 METHODOLOGY

In this section, we explain our approach for developing an Android malware detection system based on dynamic analysis using interpretable machine learning.

3.1 Data Collection and Pre-Processing

We used the *AndroZoo* dataset (Allix et al., 2016) for obtaining our goodware and part of our malware. *AndroZoo* is commonly used by research community for Android malware detection and analysis. Note that *AndroZoo* does not explicitly label apps as malware. Instead, it provides the number of antiviruses from *VirusTotal* (VirusTotal, 2023) that flagged the app as malware. Particularly, there is a property for an app called *vt_detection* in *AndroZoo*. A non-zero value for this property means that one or more antivirus tools have flagged this app as malware; otherwise, the app is considered to be goodware (*vt_detection*=0). To collect our malware samples, we chose the apps such that their *vt_detection* is more than 4 (to have more confidence in their label). A similar approach has been employed in (Pendlebury et al., 2019). Moreover, we also downloaded malware from *VirusShare* (VirusShare, 2023), which is a repository of malware samples. The quality of the dataset is of crucial importance for Android malware ML-based detection. If the dataset is not up-to-date or not representative of the population under study, the conclusions may be invalid. Therefore, while selecting the apps, we considered addressing issues including realistic class ratio and temporal inconsistency.

3.1.1 Realistic Class Ratio and Addressing Class Imbalance

To have realistic settings, the class ratio between benign and malware classes at test time is chosen similar to the one at deployment time. In the Android app domain, malware is the minority class. Similar to (Pendlebury et al., 2019; Miranda et al., 2022), we used the average overall ratio of 10% Android malware in our test samples to represent the real-world scenario. Moreover, we addressed the dataset imbalance issue by using class weights and undersampling techniques. More details are provided in Section 4.

3.1.2 Addressing Temporal Inconsistency

Temporal inconsistency (i.e., when malware and benign apps are selected randomly from different time periods)(Miranda et al., 2022; Cavallaro et al., 2023) unrealistically improves the performance of the ML-based malware detection approaches. Additionally, explanations of ML-based Android malware detection suggest that the models can accurately predict malware by relying on temporal-related features rather than the actual characteristics of malicious and benign behaviors (Liu et al., 2022b). To prevent temporal inconsistency, we chose both malware and goodware from the same timeframe, i.e., 2022.

3.2 Feature Selection

There are different features that may be included in developing a dynamic analysis solution. Many of the primary studies on dynamic analysis selected features associated with system calls and dynamic activities such as network access and memory dump to capture malicious behavior (Razgallah et al., 2021; Liu et al., 2022a; Saneeha Khalid, 2022). Among such features, system calls are the mainstream and have been favored by the research community for dynamic analysis of Android malware. In contrast to API calls, Linux kernel system calls are independent from the Android Operating System (OS) version, rendering them more robust against strategies employed by malware to evade detection. Therefore, in this paper, we focus primarily on the analysis of *system calls*. All apps interact with the platform where they are executed by requesting services through a number of available system calls. These calls define an interface that allow apps to read/write files, send/receive data through the network, read data from a sensor, make a phone call, etc. Legitimate apps can be characterized by the way they use such an interface, which facilitates the identification of malicious components in-

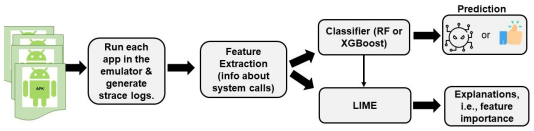


Figure 1: Proposed framework.

serted into a seemingly harmless app and, more generally, other forms of malware.

3.3 Feature Extraction Process

We explored a variety of tools to collect system calls produced by running an Android app in an emulator. We mainly used **adb** (ADB, 2023), **Android Emulator** (Android Studio, 2023), and **strace** (strace, 2023). Moreover, **Monkey** (monkey, 2023) was employed to simulate random user interactions with an app. We have examined all of the traces from our tests and we focus on extracting system calls and their frequencies. Certain system calls might be indicative of malicious behavior, such as trying to access sensitive data or performing unauthorized action. Also, unusual or high frequencies of certain system calls might indicate that it is performing some action excessively, which could be an indication of malicious activity. Thus, *we identified a list of distinctive system calls used in all the strace logs for malware and goodware. For each system call, its frequency is included in our feature vector.*

3.4 Classification Models and Explanation Method

We have opted to employ Random Forest (RF) (Breiman, 2001) and eXtreme Gradient Boost (XGBoost) (Chen et al., 2015), as our classification methods. Both of these methods are considered as ensemble learning techniques. Such techniques combine the predictions of multiple ML models. To interpret the prediction results from the above classifiers, we employed Local Interpretable Model-agnostic Explanations (LIME) (Ribeiro et al., 2016), which is a widely used approach in various other fields (Belle and Papantonis, 2021). Apart from its model-agnostic nature and the ability to offer localized interpretability, LIME is favored for its simple implementation and lower computational overhead when compared to similar techniques in the field of eXplainable Artificial Intelligence (XAI) (Dwivedi et al., 2023). Moreover, LIME has been used for explaining the ML-based predictions for static analysis of Android malware (Fan et al., 2020). LIME operates by altering specific feature values within a single data sample and then monitors the resulting impact on the output, essentially acquiring the classification label of the modified sample. Once predictions are created for a collection of perturbed input samples, LIME proceeds to approximate the decision boundary and calculate the weights that denote the significance of individual features. It should be noted that LIME is a post hoc explainable method, i.e., it is applied to a model after training. Figure 1 shows our proposed framework.

Table 1: Scenarios.

Scenario	No. Goodware	No. Malware
S1-10	2430	270
S2-15	2295	405
S3-20	2160	540

Table 2: Model performance evaluation metrics and their computed values for the first three scenarios.

Scenario	Classifier	Precision	Recall	F1-Score
S1-10	RF	1	0.33	0.5
S1-10	XGBoost	0.79	0.50	0.61
S2-15	RF	0.93	0.47	0.62
S2-15	XGBoost	0.81	0.57	0.67
S3-20	RF	0.89	0.53	0.67
S3-20	XGBoost	0.78	0.60	0.68

4 PERFORMANCE EVALUATION AND ANALYSIS OF RESULTS

4.1 Experimental Setup

We carried out our experiments on a Tensorbook with Ubuntu 22.04 OS, Intel Core i7-processor (14 cores), NVIDIA RTX 3070 Ti GPU, and 32 GB memory (DDR5-4800). We randomly chose 2,798 benign apps *AndroZoo* and 624 malware from *AndroZoo* and *VirusShare*. The Android emulator was used to emulate a Pixel 6 (API 31) device. Moreover, 146 unique system calls were identified by analyzing the traces for benign apps and malware; these have been incorporated in our feature vector. Finally, the classification algorithms are implemented using Scikit-Learn ML library (scikit-learn, 2023).

4.2 Performance Evaluation

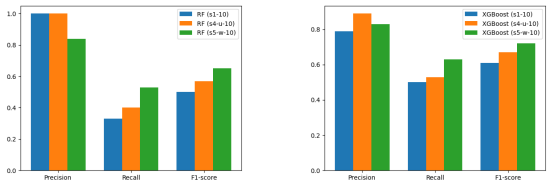
We used the grid searching method with 10-fold cross validation for hyperparameter tuning. Since our dataset is imbalanced, having high accuracy does not necessarily mean high detection rate for our positive class (i.e., malware). Therefore, we have looked into other metrics which are better representatives for the performance of our classifiers as follows.

In our first set of experiments, we examined three

scenarios, each maintaining a consistent 10% malware ratio during testing, while the training data contained 10% (S1-10), 15% (S2-15), and 20% (S3-20) malware in these respective scenarios. Table 1 shows the distribution of goodware and malware in the training sets of our scenarios. Note that although maintaining a realistic class ratio during training can reduce the overall error rate, different ratios can be employed to adjust the decision boundary, thereby balancing the trade-off between false positives and false negatives (and subsequently affect precision and recall). Table 2 summarizes the obtained values of performance metrics from the three scenarios. If we look at the recall values of the malware class, XGBoost has a higher value in all three scenarios. Therefore, XGBoost performs better than RF in terms of detecting malware (i.e., it correctly identifies malware with higher probability). It is evident from the three scenarios that When the ratio of malware in the training set goes up, the decision boundary moves towards the goodware class, i.e., the recall is increased for malware but the precision is reduced. Therefore, if our objective is to improve malware detection (i.e., recall), we can increase the malware ratio in the training set. Overall, XGBoost has a better performance compared with RF classifier due to its higher F1-score. Also, we can see the increase in F1-score when we increase the malware ratio (having a more balanced dataset) in s2-15 and s3-20 compared with s1-10 scenario. This results from the fact that the F1-score is a measure of harmonic mean of precision and recall. So, the closer the values of precision and recall are to each other, the better (higher) the F1-score will be.

Addressing Class Imbalance Using Under-Sampling: In this scenario (named s4-u-10), we applied Tomek links under-sampling technique (Haibo and Yunqian, 2013) to our training set. Tomek links consist of pairs of closely located instances belonging to different classes. Tomek links are present when two samples are each other's nearest neighbors. By eliminating the majority class instances in each pair, the separation between the two classes is widened, thus facilitating the classification process. Figure 2 shows the results of this scenario for different classifiers and in comparison with s1-10 (as our base scenario). Figure 2a displays 21%, and 14% improvement in recall and F1-score, respectively for the RF classifier. Moreover, Figure 2b illustrates 12.6% , 6%, and 9.8% improvement in precision, recall and F1-score, respectively for the XGBoost classifier.

Addressing Class Imbalance Using Class Weights: In this scenario, named s5-w-10, we used class weights to address the class imbalance problem. By



(a) (b)

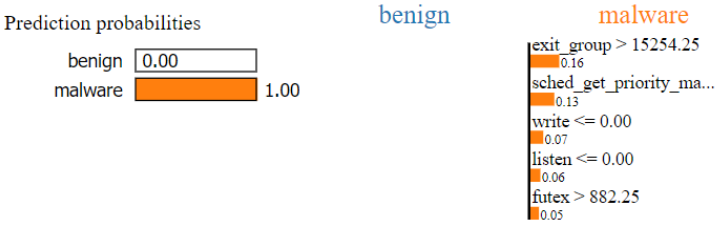
Figure 2: Performance comparison of s1-10 scenario with scenarios addressing class imbalance.

assigning higher weights to the minority class (i.e., malware class), the model is enabled to give more importance to malware samples during training and mitigate the bias towards the majority class (i.e., benign class). The class weights are inversely proportional to their frequency of occurrence in the dataset. Figure 2 illustrates the performance of the classifiers when adding the class weights. In Figure 2a, we observe 60% and 30% improvement in recall and F1-score, respectively for the RF classifier. Similarly, figure 2b shows 26% and 18% improvement in recall and F1-score, respectively for the XGBoost classifier. It should be noted that adding class weights is only effective (i.e., the improvement in the metrics is noticeable) if the dataset is highly imbalanced (i.e., ratio of 10% or less for malware in our experiments).

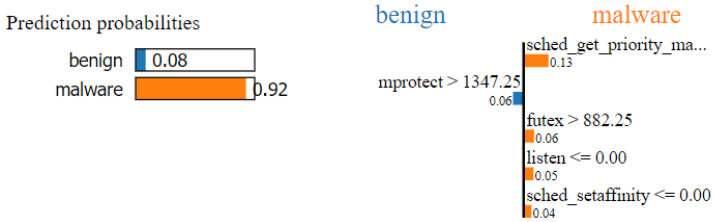
Considering the two techniques that we used for addressing the class imbalance and improving the performance of models, class weight shows more promise in terms of improving metrics when making a comparison with Tomek links technique (s4-u-10). Particularly, when adding class weights, the amount of increase in F1-score is 16% for the RF classifier and 8.2% for the XGBoost classifier compared with the Tomek links scenario (s4-u-10). A drawback to under-sampling is that we have eliminated information that may be valuable; however, in a large dataset, this may become less of a concern as the number of removed benign samples is much lower than the total number of such samples.

4.3 Analysis of LIME Explanations

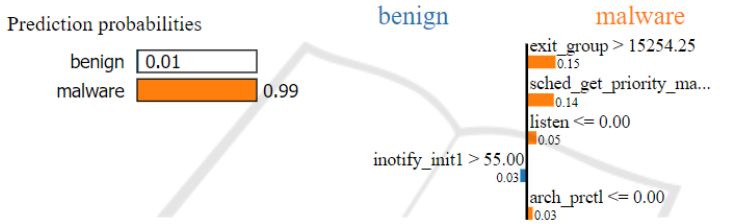
So far, our focus has been on analyzing the performance of our classifiers and mitigating the class imbalance issue for dynamic analysis. We do not have any information about the contributions of different features (i.e., the frequency of system calls) to the predictions made by our classifiers. In other words there is an important question to answer *"Why is an Android App classified as malware by our ML models?"*. To provide some insights for the prediction results, we have utilized the LIME explainable technique. Furthermore, we focus on XGBoost classifier



(a) APK package name: com.jiccos.jicco



(b) APK package name: com.wGuitarProTabs



(c) APK package name: com.hmsqw.customer

Figure 3: Examples of LIME explanations for three true positives in the testing set.

results with class weights (scenario s5-w-10) since it outperforms the RF classifier.

We analyzed the true positives (15 samples) for our testing set, which includes a total of 300 Android apps out of which 30 apps are malware. As we mentioned earlier, LIME provides local explainability, i.e., it indicates the most contributing features to the prediction result for a test instance. We configure LIME to show the top five features for a prediction in our experiments. Figure 3 shows output examples of applying LIME to three test instances. By looking at prediction probabilities, Figure 3a shows that the XGBoost classifier is 100% confident that com.jiccos.jicco is malware. Similarly, the confidence of the classifier is 92% and 99% when detecting malware in Figure 3b and Figure 3c. Moreover, the top five system calls (based on their frequencies) are shown in Figure 3. For instance, the most important system calls (based on their frequencies) to the prediction in Figure 3a are *exit_group*, *sched_get_priority_max*, *futex*, *write*, and *listen* with weights of 0.16, 0.13, 0.05, 0.07, and 0.06, respectively. Note that the descriptions of system calls can be found in (syscalls(2)–Linux manual page, 2023). Additionally, the explanation provides the reasons why the model make this prediction. Table 3 displays the top four system calls that are present in most of the true positives based on the output of LIME.

Table 3: Top system calls and their presence in true positives.

System call	Presence in true positives
exit_group	66%
sched_get_priority_max	46%
sched_getscheduler	33%
futex	26%

Explanation Quality: One important aspect of explanations to consider is their stability, i.e., the changes in explanations when utilizing the explainable method (in our case LIME) multiple times under the same conditions. We use two stability metrics, which were defined in (Visani et al., 2022) for LIME, namely Variables Stability Index (VSI) and Coefficients Stability Index (CSI). High VSI values ensure that the variables (features) obtained in various LIME instances are consistently identical. Conversely, low values indicate unreliable explanations, as different LIME calls may yield entirely different variables to explain the same machine learning decision. Regarding CSI, high values guarantee the reliability of LIME coefficients for each feature. Conversely, low values

advise practitioners to exercise caution: for a given feature, the initial LIME call may yield a specific coefficient value, but subsequent calls are likely to produce different values. As the coefficient represents the impact of the feature on the ML model decision, obtaining different values corresponds to significantly distinct explanations. The average of VSI values is 90.6% with standard deviation of 5% whereas the average of CSI values is 97.6% with standard deviation of 3.8%. Therefore, in our experiments for malware detection LIME results are more stable w.r.t. to CSI rather than VSI.

Limitations of Our Approach: It should be noted that system calls are inherently low-level and lack the extensive information found in the more semantically enriched Android APIs. Therefore, mapping them directly to the malware's behavior is a challenging task. A dataset that mapped high-level Android APIs to different representations of low-level system call traces was created in (Nisi et al., 2019). But the dataset is not accessible and we could not use it to map the system calls to API calls and subsequently relate them to an app's behavior. Moreover, we examined the reports from *VirusTotal* for some of our malware in the testing set but the features listed are mainly related to static analysis of those apps.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented an interpretable ML-based dynamic analysis approach for Android malware detection. We focused on extracting the frequency of system calls as features to train our ML models. We applied LIME to the prediction results of the ML models to indicate the most contributing features to each prediction. Finally, we evaluated the stability of LIME explanations for true positives of our model. For facilitating the reproducibility of our research, all the codes and data are publicly accessible. Some of the directions for future work are as follows. One can use an enhanced user interaction simulator such as ARES (Romdhana et al., 2022), rather than relying on the random interactions generated by *Monkey*. Moreover, using other interpretable ML techniques and comparing their explanation quality with LIME is a promising area that needs further exploration.

REFERENCES

ADB (2023). https://https://developer.android.com/tools/adb. Accessed: July 2023.

Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA. ACM.

Android Studio (2023). https://developer.android.com/studio/\\run/emulator. Accessed: July 2023.

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., and Rieck, K. (2022). Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3971–3988.

Asma Razagallah, Raphael Khoury, J.-B. P. (2022). Twindroid: a dataset of android app system call traces and trace generation pipeline. In *Proceedings of the 19th International Conference on Mining Software Repositories*.

Belle, V. and Papantonis, I. (2021). Principles and practice of explainable machine learning. *Frontiers in big Data*, page 39.

Breiman, L. (2001). Random forests. *Machine learning*, 45:5–32.

Cavallaro, L., Kinder, J., Pendlebury, F., and Pierazzi, F. (2023). Are machine learning models for malware detection ready for prime time? *IEEE Security & Privacy*, 21(2):53–56.

Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., Chen, K., Mitchell, R., Cano, I., Zhou, T., et al. (2015). Xgboost: extreme gradient boosting. *R package version 0.4-2*, 1(4):1–4.

Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., and Cavallaro, L. (2016). Droidscribe: Classifying android malware based on runtime behavior. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 252–261. IEEE.

De Lorenzo, A., Martinelli, F., Medvet, E., Mercaldo, F., and Santone, A. (2020). Visualizing the outcome of dynamic analysis of android malware with vizmal. *Journal of Information Security and Applications*, 50:102423.

Dimjašević, M., Atzeni, S., Ugrina, I., and Rakamaric, Z. (2016). Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8.

Dwivedi, R., Dave, D., Naik, H., Singhal, S., Omer, R., Patel, P., Qian, B., Wen, Z., Shah, T., Morgan, G., et al. (2023). Explainable ai (xai): Core ideas, techniques, and solutions. *ACM Computing Surveys*, 55(9):1–33.

Fan, M., Wei, W., Xie, X., Liu, Y., Guan, X., and Liu, T. (2020). Can we trust your explanations? sanity checks for interpreters in android malware analysis. *IEEE Transactions on Information Forensics and Security*, 16:838–853.

Haibo, H. and Yunqian, M. (2013). Imbalanced learning: foundations, algorithms, and applications. *Wiley-IEEE Press*, 1(27):12.

Hou, S., Saas, A., Chen, L., and Ye, Y. (2016). Deep4maldroid: A deep learning framework for an-

droid malware detection based on linux kernel system call graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, pages 104–111. IEEE.

Iqbal, S. and Zulkernine, M. (2018). Spydroid: A framework for employing multiple real-time malware detectors on android. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–8. IEEE.

Kapersky (2023). Android mobile security threats. https://www.kaspersky.com/resource-center/threats/mobile. Accessed: July 2023.

Kovacs, ByEduard (2023). New samsung message guard protects mobile devices against zero- click exploits. https://www.securityweek.com/new-samsung-message-guard-protects-mobile-devices-against-zero-click-exploits/. Accessed: July 2023.

Liu, Y., Tantithamthavorn, C., Li, L., and Liu, Y. (2022a). Deep learning for android malware defenses: a systematic literature review. *ACM Computing Surveys*, 55(8):1–36.

Liu, Y., Tantithamthavorn, C., Li, L., and Liu, Y. (2022b). Explainable ai for android malware detection: Towards understanding why the models perform so well? In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 169–180. IEEE.

Mahdavifar, S., Kadir, A. F. A., Fatemi, R., Alhadidi, D., and Ghorbani, A. A. (2020). 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)*, pages 515–522. IEEE.

Miranda, T. C., Gimenez, P.-F., Lalande, J.-F., Tong, V. V. T., and Wilke, P. (2022). Debiasing android malware datasets: How can i trust your results if your dataset is biased? *IEEE Transactions on Information Forensics and Security*, 17:2182–2197.

monkey (2023). https://developer.android.com/studio/test/other-testing-tools/monkey. Accessed: July 2023.

Nisi, D., Bianchi, A., and Fratantonio, Y. (2019). Exploring {Syscall-Based} semantics reconstruction of android applications. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 517–531.

Pan, Y., Ge, X., Fang, C., and Fan, Y. (2020). A systematic literature review of android malware detection using static analysis. *IEEE Access*, 8:116363–116379.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., and Cavallaro, L. (2019). {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 729–746.

Razgallah, A. and Khoury, R. (2021). Behavioral classification of android applications using system calls. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, pages 43–52. IEEE.

Razgallah, A., Khoury, R., Hallé, S., and Khanmohammadi, K. (2021). A survey of malware detection in android apps: Recommendations and perspectives for future research. *Computer Science Review*, 39:100358.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144.

Romdhana, A., Merlo, A., Ceccato, M., and Tonella, P. (2022). Deep reinforcement learning for black-box testing of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–29.

Saneeha Khalid, F. B. H. (2022). Evaluating dynamic analysis features for android malware categorization. In *Proceedings of the International Wireless Communications and Mobile Computin (IWCMC)*.

scikit-learn (2023). https://scikit-learn.org/stable/. Accessed: Oct 2023.

Statista (2023). Global mobile os market share 2023. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. Accessed: July 2023.

strace (2023). https://man7.org/linux/man-pages/man1/strace.1.html. Accessed: July 2023.

syscalls(2)–Linux manual page (2023). https://man7.org/linux/man-pages/man2/syscalls.2.html. Accessed: Nov 2023.

Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *Ndss*, pages 1–15.

VirusShare (2023). https://virusshare.com/. Accessed: Nov 2023.

VirusTotal (2023). Virustotal - home. https://www.virustotal.com/gui/home/upload. Accessed: August 2023.

Visani, G., Bagli, E., Chesani, F., Poluzzi, A., and Capuzzo, D. (2022). Statistical stability indices for lime: Obtaining reliable explanations for machine learning models. *Journal of the Operational Research Society*, 73(1):91–101.

Wu, B., Chen, S., Gao, C., Fan, L., Liu, Y., Wen, W., and Lyu, M. R. (2021). Why an android app is classified as malware: Toward malware classification interpretation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–29.

Yan, P. and Yan, Z. (2018). A survey on dynamic mobile malware detection. *Software Quality Journal*, 26(3):891–919.

Zhang, X., Mathur, A., Zhao, L., Rahmat, S., Niyaz, Q., Javaid, A., and Yang, X. (2022). 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*, pages 1–9.