A Robust Malware Detection Approach for Android System Based on Ensemble Learning

Wenjia Li(✉), Juecong Cai, Zi Wang, and Sihua Cheng

Department of Computer Science, New York Institute of Technology, New York, NY 10023, USA
wli20@nyit.edu

Abstract. As the number of mobile devices which is based on the Android system continues to grow rapidly, it becomes a primary target for security exploitation through undesirable malicious apps (malware) being unwittingly downloaded, which is often due to negligent user behavior patterns that grant unnecessary permissions to malicious apps or simply malware evolving to be sophisticated enough to bypass systematic detection. There have been numerous attempts to use machine learning to capture an application's malicious behavior focusing on features deemed to be germane to high security risks, but most of them typically focus only on a single algorithm, which is not representative of a huge family of ensemble techniques. In this paper, we develop an ensemble learning based malware detection approach for the Android system. To validate the performance of the proposed approach, we have conducted some experiments on the real world Android app dataset, which contains 3618 features that are initially obtained from the static, dynamic and ICC analyses. We then select 567 important features through feature selection. The overall detection accuracy is 97.73%, accompanied by a high 97.66% F-1 score that reflects a high relationship between precision (97.06%) and recall (98.28%). The experimental results clearly show that the ensemble learning based malware detection approach could effectively identify malware for the Android system.

Keywords: Android · Security · Malware · Machine learning · Ensemble learning

1 Introduction

In the second quarter of 2021, the Android operating system is reported to have 72.84% share of the mobile operating system market, which continues to dominate the mobile OS market throughout the world [1]. Given how common it is to find an Android device today, it is without doubt that its prevalence and open source nature easily opens up security vulnerabilities, where various security exploitations are used to gain unwanted access into Android phones. While Google Play provides some level of security protection, it is still inevitable that break-ins still occur [2].

G. Wang et al. (Eds.): UbiSec 2021, CCIS 1557, pp. 309–321, 2022.
https://doi.org/10.1007/978-981-19-0468-4_23

The Android platform is known to be a permission based system [11]. The apps are required to explicitly request for a corresponding permission from the user during the installation process to perform certain tasks on the Android devices, such as sending a SMS message or gaining access to the Internet. However, many users tend to arbitrarily grant permissions to unknown Android apps without even looking at what types of permissions they are requesting, therefore significantly weakening the protocols set in place for protection provided by the Android permission system [3, 5, 13].

Apart from user problems, malware themselves have become adept at circumventing standard detection protocols such as Google Bouncer and other standard anti-intrusion gateways that Google uses as deterrence. The Google Bouncer is capable of reducing the number of malicious apps by as much as 40% [3], but its predictability makes it easy to overcome. Google Bouncer uses the generic and open source QEMU as a machine emulator [4], and only performs dynamic analysis. In 2017, Google Bouncer became part of Google Play Protect, which is a system that regularly scans Android apps for potential security threats.

There are two broad methods of bypassing the security mechanism deployed at Google Play, which are namely Delayed Attack and Update Attack [3]. Delayed Attack involves meeting Bouncer's basic criteria, meaning an app with malicious payload (i.e. Trojan) does not misbehave during dynamic analysis and pretends to be "clean" until the five-minute scan has passed. The malicious code only starts to run after the app is downloaded onto the Android device. Update Attack is even harder to detect than Delayed Attack because the app does not even have to contain any malicious component at all. The basic app is downloaded as clean, but once on a device, it starts to "update" itself with malicious code or connect to a remote server to upload stolen personal data [3]. These malicious apps can make themselves behave like legit apps and are only flagged as malicious when they activate selected functions to perform their infiltration process. As such, it becomes more difficult to identify a malware because it can cleverly disguise itself from detection by either suppressing its malicious "tendencies" or simulate benign app behavior during the detection phase, until they gain access into a user's phone.

With so many varieties of Android malware, it is fair to hypothesize for the need to have an algorithm that is capable of detecting different types of malware. However, most of the prior research efforts typically focus only on using one single algorithm to detect malware, which may not work well on various types of Android malware. By applying the concept of ensemble learning, there could be many different ways in which one can effectively combine those algorithms together to achieve better detection results. Thus, it makes more sense to find the best algorithm that brings the most optimized results to solve the problem, preferring flexibility over proving the effectiveness of one single algorithm [12].

Ensemble learning is chosen because it has a proven track record in many fields [15], and more importantly, it is flexible and customizable, allowing many combinations. Using multiple learning algorithms tends to obtain better performance compared to any single learning algorithm, and most of the ensemble algorithms are fairly easy to implement, scale well to large datasets, and are quick to execute.

The main contributions of this paper are:

- We implement multiple ensemble algorithms at the same time for comprehensive in-depth comparison measured by different metrics such as accuracy, precision and recall. On top of that, this paper will also compare the performance of ensemble learning algorithms with those of individual machine learning algorithms to illustrate the effectiveness of the ensemble algorithms.
- We deploy multiple algorithms as a basket of models with diverse applications, and then combine the accuracy scores using an ensemble voting algorithm to boost results.
- We illustrate the feasibility and performance of the proposed malware detection approach using a real world Android app dataset. Experimental results show that the ensemble learning based malware detection approach can effectively identify malware with very high accuracy.

2 Related Work

2.1 Common Techniques Used in Malware Detection

A recent survey has been conducted by Liu et al. [8]. In this work, the authors first introduce the basics of Android applications, including the Android system architecture, security mechanisms, and classification of Android malware. Then, the authors analyze and summarize the current research status of malware detection from different aspects, including sample acquisition, data pre-processing, feature selection, machine learning algorithms, and the performance evaluation.

Kouliaridis et al. [12] summarized several papers that adopted different algorithms to detect malware, and quickly noted that a singular view of using static or dynamic analysis alone have proven to be unreliable as they can be easily evaded with code obfuscation and execution-stalling techniques respectively.

2.2 General Benchmarks

First and foremost, Drebin [16] makes up an important part of this study since all our malicious apps came from the Drebin study, making it a requisite benchmark for future comparisons. Drebin is a lightweight method for Android malware detection that works directly on the smartphone like an anti-virus software and identifies suspicious apps by name and attribute at the same time. It uses broad static analysis to collect features from 123,453 apps, with 5,560 of them being malware. As a pioneer, Drebin had a performance of a 94% accuracy score with few false alarms (false positives) of 1%, and explanations were provided for each malware detected, stating their properties - thus setting a very high bar for future studies. It was tested on five popular smartphones with an average runtime of 10 s per analysis [16].

Another high performing example is Yerima et al. [15] using Ensemble Learning on malware detection, promised to be high accuracy using 179 different features from diverse categories of malware behavior, emphasizing on robustness and diversity to malware problem solving. The experiment used a total of 6,863 Android applications

that they got from McAfee's internal repository, out of which 2925 were malware and 3938 apps were benign. The best results that Yerima et al. has yielded is from combining Random Forest with Naïve Bayes, scoring an accuracy of 97.5%, with a false positive rate of around 2.3% and an AUC of 99.3%. However, our research does not use the AUC but instead calculates using the F-1 score to decide the relationship between precision and recall. Random Forest is used because randomness provides diversity in samples and robustness in speed of execution without having too much data processing. Interestingly, the Yerima et al. features did not exactly come from diverse sources as the study claimed, because it only made use of static analysis from permissions and API-calls. Judging from the results, it does seem like just crawling features from these two areas are more than enough.

Yang et al. came up with DroidMiner [17] that implemented one ensemble learning algorithm, namely the Random Forest algorithm, comparing its effectiveness against Naïve Bayes, Support Vector Machine (SVM), and Decision Tree. They evaluated 2400 malicious apps out of a corpus of over 77,000 apps, made up of 67,000 third party apps and 10,000 from Google Play. DroidMiner achieved the highest accuracy achieved on Random Forest with 95.3%. The other scores are 82.2% for Naïve Bayes, 86.7% for SVM and 92.4% for Decision Tree.

2.3 Collecting Features

Wang et al. came up with the top 40 most risky permissions [19], where the team had analyzed the risk of individual permissions and collaborative permissions by groups using machine learning techniques, and then performed feature ranking on them. Wang et al. reasoned that although Android enforces restrictions through a dual-party system, the Android system does not always require full declaration of permissions, while users are not always aware of the exact purpose of granting access to certain app functions [10, 19]. Apps can end up requesting for unnecessary permissions, resulting in over-privileged applications that often leave security loopholes that malware can quickly exploit. The results were classified using the Support Vector Machine (SVM), Decision Tree, and Random Forest techniques [19]. In addition, there were also other malware detection approaches which used various machine learning and deep learning algorithms based on permissions, API calls and other internal app features [30–37].

Comar et al. warned that for new-generation malware, the static detection method is no longer effective, known as zero-day malware [13, 14]. A zero-day malware (also known as next-generation malware) is a previously unknown malware for which specific antivirus software signatures are still unavailable. It is a vulnerability in software not known to the vendor, which can be exploited by hackers before the vendor becomes aware and patches a solution to fix it [14]. Therefore, apart from permission features, efforts should be made to venture into other avenues of features collection such as dynamic, ICC analysis, or other out-of-box areas as well.

Instead of simply going for inherent features of Android apps like permissions, API-calls, CPU usage and system calls, Munoz et al. [10] focused their study on identifying what they call indirect features or meta-data, mentioning that Google Play is also a fantastic repository of information for helping detect a malware, focusing on Application Category, Developer-related, Certificate-related features, and Social-related features.

Munoz et al. [10] collected a total of 48 features, and used Logistic Regression as their classifier, with the results showing that social-related features are not very useful. Some of the features categories showed high accuracy scores, but the precision, recall and F-1 are lackluster. On the other hand, developer-related and certificate-related features did show good promise. The study managed to greatly reduce their false negative rate but at the expense of their false positive rate, which they did not publish figures for.

2.4 ICC Analysis Using Intents and Intent-Filters

Apart from permissions, the Android system also uses intent and intent-filters as a mechanism for inter-process communication (ICC) between functions. The intent mechanism is predominantly for starting an activity, a service or sending a broadcast [6]. Explicit intents are normally safe and straightforward in its intention to access activities on an app, while implicit intents are not recommended as it contains inherent unsafe automatic app usage associations tied to them that may bypass the users' knowledge of app activity. Malicious apps can take advantage of this vulnerability to gain access to high security permissions through intent interception or intent spoofing.

Xu et al. [9] came up with the idea to trace malware by capturing ICC usage data from apps, and created a program named ICCDetector, which can tackle the problem of false positive and negative rates. They noted that Kirin [28] detects malwares by matching their required permissions against pre-defined security rules, while both DroidMiner [17] and DroidAPIMiner [7] build malware detection models based on API-related features. The weakness of these methods is that they treat the detected applications as standalone entities in Android platforms, assuming that the Android OS will keep separate apps mutually exclusive from each other, when in actual fact, resources could be shared through ICC means that is often overlooked.

3 Ensemble Learning Based Malware Detection for Android

3.1 System Architecture

The overall system architecture is shown as in Fig. 1.

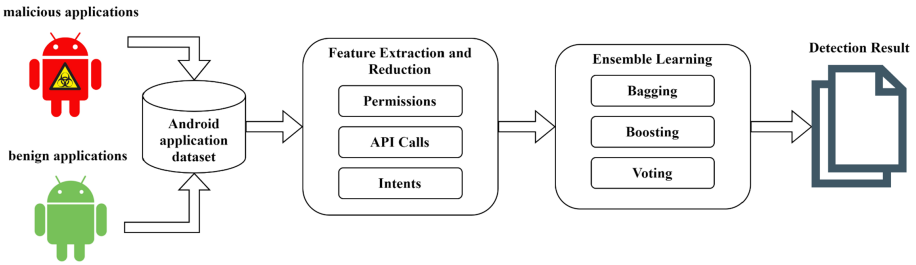

Fig. 1. Overall system architecture

In this work, we adopt the bucket of models [23] concept where a basket of algorithms is put together, the results are voted, and among all the accuracy scores, the best

performing algorithm is chosen. The algorithms that we use in ensemble learning could be classified into two broad categories, one being ensemble learning models for bagging and boosting, and the other is a group of widely used machine learning algorithms such as Support Vector Machine (SVM), k-nearest neighbors (k-NN), etc. All of them will be put to a vote, which is also an ensemble algorithm, and all the algorithms, including the voting ensembles are sorted, and the top performing algorithm is chosen to ensure optimized results.

3.2 Decision Tree

We will first address the Decision Tree learning algorithm [20] as it is the basic function for most of the ensuing Ensemble Learning algorithms. Decision Tree generally uses greedy search as its searching strategy [20]. It has two criteria by which to split a tree to derive its results, namely the Gini Impurity or Entropy. By default, Python operates the Decision Tree classifier using the Gini Impurity if no further instructions are given. Using either does not seem to make a big difference in result in this particular study, as the accuracy scores of both criteria based on the dataset of this study reap about the results, with a difference of about 0.5%, which is insignificant.

The Gini Impurity [20] is a measure of the frequency of misclassified labels of randomly chosen samples in the branch. It is used to compute the impurities present in the partition dataset. It is defined as follows.

$$I_g(f) = \sum_{i=1}^{j} f_i(1-f_i) = \sum_{i=1}^{j} \left(f_i - f_i^2\right) = \sum_{i=1}^{j} f_i - \sum_{i=1}^{j} f_i^2 = 1 - \sum_{i=1}^{j} f_i^2 = \sum_{i \neq k} f_i f_k \tag{1}$$

Where f_i is the probability of an item with the label i being randomly chosen, while $(1-f_i)$ is the probability of the labeling being a mistake. The Gini Impurity is the sum of f_i multiplied by $(1-f_i)$, with i starting at 1 all the way till J, where J is the total number of classes or features within the set.

Entropy [21] is for calculating information gain, where it is defined as follows.

$$H(T) = I_E(p_1, p_2, ...p_n) = -\sum_{i=1}^{J} p_i log_2 \boldsymbol{p_i} \tag{2}$$

Here, p_1, p_2,..., p_n represent the percentage of each class present in the child node that results from a split in the tree, and all the percentage ultimately add up to 100% or the fraction value of 1.

Information gain [21] is calculated as the difference of the entropy values from where the tree splits due to a reduction of entropy until values become homogeneous, when no more information can be gained. It is represented in the formula below.

$$IG(T, a) = H(T) - H(T/a) \tag{3}$$

It is important to note that the Decision Tree is not without limitations [20, 21]. It has problems expressing hard-to-learn concepts such as XOR, parity or multiplexer problems [22]. It also tends to be biased when using Entropy for information gain calculations, as

it tends to favor attributes with more levels. It is very data sensitive, where a small change in the dataset can result in big changes in how the tree splits and therefore affecting the final prediction. Greedy search strategy is the Decision Tree's biggest flaw as it takes the most "convenient" close-by result and hence tends to return a favorable optimal local result but does not do a thorough and complete clean search through the entire tree, and hence does not always return the best optimal general result for the whole tree.

Due to this shortcoming of the Decision Tree algorithm, its accuracy and predictivity often suffers, especially when too many classes or features are involved, as the Decision Tree algorithm performs best when the trees are small. As such, it can have a problem of overfitting data. However, as mentioned, it works great as a base skeleton, and this study will build on the Decision Tree to improve its predictive via enhancements using various Ensemble Learning techniques. The three most popular methods for combining the predictions from different models currently are Bagging, Boosting and Voting.

3.3 Bagging

Bagging is fully known as Bootstrap Aggregation, which generally involves taking multiple samples from the training dataset and then training a model from each sample. Bagging generates multiple models using the same algorithm, using random sub-samples of the dataset drawn using the bootstrap sampling method from the original dataset, where some original examples may appear more than once and some not present in the sample. The final output prediction is averaged across the predictions of all of the sub-models.

Bagging performs best with base algorithms with high variance, and is excellent at reducing variance, thus stabilizing and improving the predictive performance. Unlike Decision Tree, Bagging uses more than one tree, and the user can either specify the maximum number of trees, or the algorithm will keep running until it runs off branches to split into. The samples of the training dataset are taken with replacement, which means the object is put back into the bag so that the number of samples to choose from is the same for every draw when constructing the model.

The three bagging models studied in this research are described as follows [18].

Bagged Decision Trees. The various bagging models are actually very similar to each other, and each can be said to be a level-up improvement of the other. Bagged Decision Trees is the most basic of the method. As its name suggests, it uses a Decision Tree as a base-estimator [18], and then does multi-sampling and can have multiple trees, and the results were combined using averaging to overcome the shortcoming of using a single tree. Decision trees that grow very deep tend to learn highly irregular patterns, and overfit their training sets. They may have low bias but end up with very high variance due to noise in the training data. Bagging is thus great for variance reduction without raising the bias.

Random Forest. Random forest [24] is an extension of bagged decision trees, and it does not train greedily when choosing the best split point in the construction of the tree, instead a random subset of features is considered for each split. This is also known as "feature bagging", and features that are deemed as strong predictors for output classification will be repeatedly chosen by most trees, thus causing correlations between trees.

Random Forest can deal with large numbers of training instances, missing values, and irrelevant features without running into problems. Random Forest deals with more than one tree and has multiple models. It reduces variance by averaging the predictions of a set of m trees with individual weight functions W_j, and can be represented by the following prediction function, which is similar to the K-Nearest Neighbor (KNN) algorithm but taking into account the number of trees and features used in the Random Forest [24, 25].

Extra Trees. Extra Trees does not mean that the algorithm involves using even more trees, but rather takes randomization up one notch from Random Forest. Instead of using the Gini Impurity, Entropy or feature calculation is used to decide a split in the tree, Extra Trees selects a random value for each feature under consideration based on the bootstrap sample, which is random sampling with replacement.

3.4 Boosting

The purpose of boosting algorithms is mainly for reducing bias and variance and turning weak learner algorithms into strong ones [26]. It achieves this by creating a sequence of models that attempt to correct mistakes of the models that came before in the sequence. Each model makes predictions which are weighted by their accuracy and results are combined to produce a final output prediction. The sequence starts with weights that are assigned according to the weak learner's accuracy. The weights are re-adjusted as misclassified samples gain weight and correct classifications lose weight, helping the algorithm to "learn its mistakes".

The two main algorithms often used in boosting techniques are Gradient Tree Boosting and AdaBoost, which are described further below.

Adaboost. AdaBoost, or Adaptive Boosting, begins by fitting a weak learner classifier on the original dataset and then fits additional copies of the same classifier on the same dataset, adjusting the weights of misclassified instances so that subsequent classifiers focus more on difficult cases.

Gradient Tree Boosting. Gradient Tree Boosting or Gradient Boosted Regression Trees (GBRT) is an accurate and effective off-the-shelf approach and is usually used with decision trees of a fixed size as base learners. It builds as an additive model in a forward stage-wise fashion, and instead of learning from errors like AdaBoost, GBRT optimizes its cost function by iteratively choosing samples that point in the negative gradient direction. When dealing with regression trees, the GBRT is fit on the negative gradient of the binomial or multinomial deviance loss function, using logistic regression as loss function. On the other hand, it recovers the Adaboost algorithm for exponential loss function.

3.5 Voting

Unlike Bagging and Boosting that build multiple models using the same algorithm, voting combines the results from multiple algorithms where the majority vote wins. It has the properties of error correction and predictive boosting.

There are generally two kinds of voting: the first is known as "Democratic Voting", where all members in the vote are assigned equal weights, and the other is "Weighted Voting" where significant members that are better performers are given more weights than poorer performers so that the predictive value will lean towards a classification that tends to choose the more "correct" answer. Both voting techniques are generally fine, but Kaggle [27] recommends using "3 Best vs the Rest" where higher weights are assigned to the top three performers compared to the rest, and the results are generally slightly better than the Democratic Voting technique.

4 Experimental Study

The experimental dataset consists of a total of 3618 features extracted from 4430 Android applications, with a 50% malicious and 50% benign split. The malicious apps were mostly from Drebin [16], while benign apps were directly downloaded from Google Play. The features come from various sources such as permissions, API calls, system calls (dynamic analysis) and ICC (inter-component communication). However, it should be noted that most dynamic analysis and ICC features have either very sparse data or have very rarely shared occurrences between apps.

The Extra Tree algorithm [29] has been used for feature reduction, which uses feature importance ranking scores to weed out unimportant features that have zero values or are not useful for telling apps apart. The dataset started out with 3618 features, and after feature reduction, only 567 are identified to be important.

4.1 Necessity of Using Multiple Machine Learning Algorithms

This section aims to show the necessity of using multiple algorithms over just using one. To make the experiments simple, the dataset has been randomly divided into three different sets containing different numbers of features, and each set is executed twice - once as a full dataset, and a second time after feature reduction, thereby creating 6 different scenarios to compare. Since the full dataset took a longer time to execute, the execution time has been included to show how long it takes, and the result is about the same as the one after feature reduction.

From Fig. 2, it can be clearly seen that from just using different numbers of features alone, the algorithm that has the highest accuracy differs, which indicates that one single algorithm is not always the best choice for every scenario. Thus, it is necessary to implement a basket of models using ensemble learning techniques.

In addition, we also observe that the full dataset with 3618 features has an exaggeratingly long execution time at 26745.15 s (about 7.4 h), which is a huge trade-off for the 1% increase in accuracy from the dataset with 800 features, so more features do not always help improve the performance. On the other hand, using feature reduction together with our basket of models allows us an 84.8 times improvement (from 26745.15 s to 315.43 s) in execution time efficiency, achieving roughly the same accuracy at 97.58%. Therefore, it is feasible to apply the ensemble learning technique, which can achieve similar accuracy with much less time overhead.

Feature Set	Best Algorithm & Accuracy	Exec. Time	Reduced Feature Set	Best Algorithm & Accuracy	Exec. Time
228 features	Gradient Boosting: 95.27%	40.29s	**54 features**	Extra Tree: 95.17%	48.94s
800 features	Weighted Voting: 96.62%	244.69s	**151 features**	Extra Tree: 96.84%	93.59s
3618 features	Extra Tree: 97.62%	26745.15s	**567 features**	Weighted Voting: 97.58%	315.43s

Fig. 2. Execution results by dataset

4.2 Performance Comparison of Different Algorithms

As shown in Fig. 3, all the algorithms listed in our basket of models have been applied to the dataset, and the results are sorted by their accuracy scores in descending order. It can be easily found that all the top performers are Ensemble Learning based approaches. The only exception is AdaBoost which falls behind Logistic Regression (which is not an ensemble learning based algorithm). Still, it performs better than most other non-ensemble algorithms. Therefore, we can conclude that ensemble learning based algorithms are the overall winners here.

At 97.73% accuracy, Weighted Voting is the best performing algorithm, while also boasting the highest recall score at 98.28% which is excellent with only 1.66% false negative rate. This means it managed to identify most of the malicious apps available in the dataset despite the fact that it contains members in the vote with low scores, but since the votes are weighted, chances are the weak votes have been compensated by the stronger votes, and hence being resilient to those errors.

3618 features, 567 reduced						
Algorithm	FN 10	FP 01	Accuracy	Precision	Recall	F1
VotingWeighted	1.66%	2.87%	97.73%	97.06%	98.28%	97.66%
VotingDemocracy	2.44%	2.64%	97.46%	97.26%	97.47%	97.37%
ExtraTree	2.88%	2.53%	97.30%	97.37%	97.01%	97.19%
GradientTreeBoosting	1.88%	3.79%	97.16%	96.14%	98.05%	97.09%
RandomForest	2.77%	3.10%	97.06%	96.79%	97.13%	96.96%
BaggedDecisionTree	2.77%	3.22%	97.01%	96.68%	97.13%	96.90%
LogisticRegression (Linear)	3.88%	3.79%	96.17%	96.06%	95.98%	96.02%
AdaBoost	5.43%	3.79%	95.39%	96.00%	94.37%	95.18%
SVC Radial	6.87%	2.87%	95.13%	96.89%	92.87%	94.84%
DecisionTreeEntropy	4.99%	6.55%	94.23%	93.32%	94.83%	94.07%
K Nearest Neighbour	6.87%	5.29%	93.92%	94.43%	92.87%	93.64%
DecisionTreeGini	5.43%	6.78%	93.90%	93.07%	94.37%	93.71%
GaussianNaiveBayesB	3.22%	39.20%	78.81%	70.40%	96.67%	81.47%

Fig. 3. Performance of ensemble learning based approach vs. single algorithm

5 Conclusion

In this paper, we propose a robust malware detection approach based on the ensemble learning technique. The experimental study showed that it can achieve high accuracy, recall and precision. Moreover, it is most interesting to note that through the experiments, it turns out that despite the boosting and error correcting nature of the majority voting based algorithms, they do not always guarantee the top results. They may be within the top 3–5 best performing algorithms, but depending on the dataset, bagging algorithms like Random Forest or Extra Trees still prevail, and Gradient Boosting on certain occasions.

As for the future direction, we would like to explore the adversarial attacks against the malware detectors, and how they could be coped with. Adversarial attacks have recently become a major threat to the malware detectors, as they could mutate or tamper with the dataset that the malware detectors are using. Consequently, the performance of the malware detectors will be severely degraded in the presence of adversarial attacks. Therefore, it would be valuable to do some research in how the adversarial attacks may impact the malware detectors, and how to address these attacks.

References

1. O'Dea, S.: Market share of mobile operating systems worldwide 2012–2021. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. Accessed 29 June 2021
2. Cisco. Midyear Security Report (2015). http://www.cisco.com/web/offers/pdfs/cisco-msr-2015.pdf
3. Trend Micro. A Look at Google Bouncer (2012). http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/
4. QEMU. 2016. http://wiki.qemu.org/Main_Page
5. Stefanko, L.: Android Trojan drops in, despite Google's Bouncer. ESET, 22 September 2015–12:48 pm (2015). http://www.welivesecurity.com/2015/09/22/android-trojan-drops-in-despite-googles-bouncer/
6. Android Developer. Intents and Intent Filters. https://developer.android.com/guide/components/intents-filters.html
7. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: mining API-level features for robust malware detection in Android. In: Proceedings of the 9th International ICST Conference on Security and Privacy in Communication Networks (Secure Comm), Sydney, NSW, Australia, September 2013, pp. 86–103 (2013). https://doi.org/10.1007/978-3-319-04283-1-6
8. Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., Liu, H.: A review of android malware detection approaches based on machine learning. IEEE Access **8**, 124579–124607 (2020)
9. Xu, K., Li, Y., Deng, R.H.: ICCDetector: ICC-based malware detection on android. IEEE Trans. Inf. Forensics Secur. **11**(6) (2016)
10. Munoz, A., Martin, I., Guzman, A., Hernandez, J.A.: Android malware detection from Google Play meta-data: selection of important features. IEEE CNS 2015 poster session (2015)
11. Android Developers. Manifest permission (2016). http://developer.android.com/reference/android/Manifest.permission.html
12. Kouliaridis, V., Kambourakis, G.: A Comprehensive survey on machine learning techniques for android malware detection. Information **12**, 185 (2021). https://doi.org/10.3390/info12050185

13. Comar, P.M., Liu, L., Saha, S., Tan, P.-N., Nucci, A.: Combining supervised and unsupervised learning for zero-day malware detection. In: Proceedings of IEEE INFOCOM 2013 (2013)
14. PC Tools, Symantec. What is a Zero-Day Vulnerability?(2010). http://www.pctools.com/security-news/zero-day-vulnerability/
15. Yerima, S.Y., Sezer, S., Muttik. I.: High accuracy android malware detection using ensemble learning. IET Inf. Secur. (2015). ISSN:1751-8717. Doi: https://doi.org/10.1049/iet-ifs.2014.0099
16. Arp, D., Spreitzenbarth, M., Huebner, M., Gascon, H., Rieck, K.: Drebin: effective and explainable detection of android malware in your pocket. In: NDSS 2014, 23–26 February 2014, Internet Society, San Diego (2014). ISBN:1-891562-35-5
17. Kutyłowski, M., Vaidya, J. (eds.): ESORICS 2014. LNCS, vol. 8712. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11203-9
18. Scikit-Learn. Ensemble learning. http://scikit-learn.org/stable/modules/ensemble.html
19. Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., Zhang, X.: Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans. Inf. Forensics Secur. **9**, 1869–1882 (2014)
20. Lior Rokach, O. Maimon, 2008. Data Mining with Decision Trees: Theory and Applications, 2nd edn. World Scientific Pub Co Inc., Singapore (2007). ISBN: 978-9812771711
21. Witten, I., Frank, E., Hall, M.: Data Mining, pp. 102–103. Morgan Kaufmann. Burlington (2011). ISBN: 9780-12-374856-0
22. Gareth, J., Witten, D., Hastie, T., Tibshirani, R.: An Introduction to Statistical Learning, p. 315. Springer, New York (2015). https://doi.org/10.1007/978-1-4614-7138-7. ISBN 978-14614-7137-0
23. Zenko, B.: Is combining classifiers better than selecting the best one. Mach. Learn. **2004**, 255–273 (2004)
24. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning, 2nd edn., Springer, New York (2008). https://doi.org/10.1007/978-0-387-84858-7. ISBN:0-387-95284-5
25. Lin, Y., Jeon, Y.: Random forests and adaptive nearest neighbors (Technical report). Technical Report No. 1055. University of Wisconsin (2002)
26. Breiman, L.: Arcing [Boosting] is more successful than bagging in variance reduction. Bias, variance, and arcing classifiers. Technical Report (1996), Accessed 19 Jan 2015
27. Kaggle. Ensembling Guide. https://mlwave.com/kaggle-ensembling-guide/
28. Enck, W., Ongtang, M., Mcdaniel, P.: On lightweight mobile phone application certification. In: ACM Conference on Computer and Communications Security, pp. 235–245 (2009)
29. Scikit-Learn, Extra Tree Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html
30. Li, W., Ge, J., Dai, G.: Detecting malware for android platform: an SVM-based approach. In: 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, pp. 464–469. IEEE (2015)
31. Wang, Z., Cai, J., Cheng, S., Li. W.: DroidDeepLearner: identifying android malware using deep learning. In: 2016 IEEE 37th Sarnoff Symposium, pp. 160–165. IEEE (2016)
32. Monica, K., Li, W.: Lightweight malware detection based on machine learning algorithms and the android manifest file. In: 2016 IEEE MIT Undergraduate Research Technology Conference (URTC), pp. 1–3. IEEE (2016)
33. Li, W., Wang, Z., Cai, J., Cheng, S.: An android malware detection approach using weight-adjusted deep learning. In: 2018 International Conference on Computing, Networking and Communications (ICNC), pp. 437–441. IEEE (2018)
34. Su, X., Liu, X., Lin, J., He, S., Zhangjie, F., Li, W.: De-cloaking malicious activities in smartphones using HTTP flow mining. KSII Trans. Internet Inf. Syst. (TIIS) **11**(6), 3230–3253 (2017)

35. Li, W., Bala, N., Ahmar, A., Tovar, F., Battu, A., Bambarkar, P.: A robust malware detection approach for android system against adversarial example attacks. In: 2019 IEEE 5th International Conference on Collaboration and Internet Computing (CIC), pp. 360–365. IEEE (2019)
36. Su, X., Xiao, L., Li, W., Liu, X., Li, K.-C., Liang, W.: DroidPortrait: android malware portrait construction based on multidimensional behavior analysis. Appl. Sci. **10**(11), 3978 (2020)
37. Bala, N., Ahmar, A., Li, W., Tovar, F., Battu, A., Bambarkar, P.: DroidEnemy: battling adversarial example attacks for Android malware detection. Digit. Commun. Netw. (2021)