A Comprehensive Study of Learning-based Android Malware Detectors under Challenging Environments

Cuiying Gao
Huazhong University of Science and Technology, China
gaocy@hust.edu.cn

Gaozhun Huang
Huazhong University of Science and Technology, China
gaozhun@hust.edu.cn

Heng Li
Huazhong University of Science and Technology, China
liheng@hust.edu.cn

Bang Wu
Huazhong University of Science and Technology, China
bangw@hust.edu.cn

Yueming Wu
Nanyang Technological University, Singapore
wuyueming21@gmail.com

Wei Yuan*
Huazhong University of Science and Technology, China
yuanwei@mail.hust.edu.cn

ABSTRACT

Recent years have witnessed the proliferation of learning-based Android malware detectors. These detectors can be categorized into three types, String-based, Image-based and Graph-based. Most of them have achieved good detection performance under the ideal setting. In reality, however, detectors often face out-of-distribution samples due to the factors such as code obfuscation, concept drift (e.g., software development technique evolution and new malware category emergence), and adversarial examples (AEs). This problem has attracted increasing attention, but there is a lack of comparative studies that evaluate the existing various types of detectors under these challenging environments. In order to fill this gap, we select 12 representative detectors from three types of detectors, and evaluate them in the challenging scenarios involving code obfuscation, concept drift and AEs, respectively. Experimental results reveal that none of the evaluated detectors can maintain their ideal-setting detection performance, and the performance of different types of detectors varies significantly under various challenging environments. We identify several factors contributing to the performance deterioration of detectors, including the limitations of feature extraction methods and learning models. We also analyze the reasons why the detectors of different types show significant performance differences when facing code obfuscation, concept drift and AEs. Finally, we provide practical suggestions from the perspectives of users and researchers, respectively. We hope our work can help understand the detectors of different types, and provide guidance for enhancing their performance and robustness.

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**.

*Corresponding authors

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623320

KEYWORDS

Android Malware Detection, Machine Learning, Code Obfuscation, Concept Drift, Adversarial Examples

ACM Reference Format:
Cuiying Gao, Gaozhun Huang, Heng Li, Bang Wu, Yueming Wu, and Wei Yuan. 2024. A Comprehensive Study of Learning-based Android Malware Detectors under Challenging Environments. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623320

1 INTRODUCTION

As the most popular mobile operating system, Android has become the main target of malware authors. According to AVTest [9], the number of Android malware has reached 33,184,323 in 2022, which is about 2405 times what it was ten years ago. This tremendous growth in malware has become a serious threat, and it is essential to take the necessary measures to stop malware spread.

In recent years, a variety of learning-based detection methods (or *detectors*) have been proposed to identify Android malware (i.e., malware detection) or classify malware into their respective categories or families [38]. To avoid executing Android apps, most of them use static features, e.g., Drebin [7] and MaMaDroid [31]. According to the types of features, these detectors can be further divided into three categories: String-based, Image-based and Graph-based. The String-based detectors extract the required string sequence (e.g. APIs and permissions [7]) from an APK, and then encode the sequence into a feature vector for malware detection. The Image-based detectors convert an APK into an image, and employ an image classification method to identify malware [51] [54]. The Graph-based detectors extract semantic information from an APK, and represent the latter as a graph (e.g., function call graph [49] [20] [50]). The graph is usually further converted into a feature vector and then fed to a classifier for prediction. Most of the existing learning-based detectors can achieve satisfactory or even extremely high detection performance when the training and test data follow an identical distribution [31] [63]. However, existing research has revealed that models often encounter *out-of-distribution* samples that will degrade their detection performance [22] [27].

As shown in Figure 1, in real world, out-of-distribution samples may be generated due to three non-negligible factors: *Code obfuscation*, *Concept drift* and *Adversarial examples (AEs)*. First, obfuscation

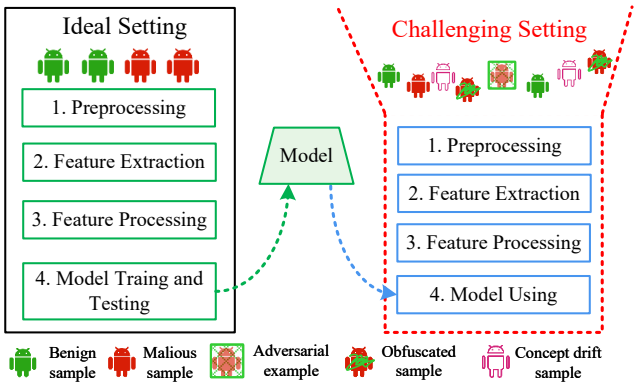


Figure 1: The detectors are usually trained under the ideal setting and used under challenging environments.

tools can produce obfuscated software with the same functionality but more complicated code [24] [21]. Second, both benign and malicious software may transform and evolve [45], and new variants of malware can emerge [36]. Third, attackers may launch adversarial attacks and produce adversarial malware that can evade detection [60] [15]. The above three factors yield out-of-distribution samples, which breaks the ideal setting assumption, and necessitates re-examining the performance of the existing malware detectors under more realistic environments.

Currently, several survey articles have summarized the existing Android malware detection methods [35] [38] [47] [43], and also highlighted the challenges that detectors face in the real world. However, they did not re-evaluate the detectors, and solely analyzed the results by the authors of detectors. Several recently proposed detectors have also undergone evaluation under challenging environments and conducted performance comparisons. For example, RevealDroid [22] was evaluated under code obfuscation and concept drift. MaMaDroid was evaluated under concept drift. Since these experiments were conducted on different datasets and under different settings, it is difficult to make a comparison among the evaluated detectors. Furthermore, some important works devote more efforts to evaluation under challenging environments. For example, TESSERACT [36] argued that the performance of existing detectors is often overestimated due to the biases in space and time, and proposed some experimental constraints for these biases. Borja et al. [34] proposed a fairer evaluation framework considering realistic factors. These works provide deeper insights on detection performance under challenging settings. However, due to the vast number and diverse categories of detectors, as well as the complexity of the real world, there are still some unclear issues: 1) how different types of detectors perform under the ideal and challenging environments; 2) why different types of detectors perform better or worse under different challenging environments; 3) what are the future directions for improving detectors and how users select detectors? Exploring these issues can further enrich the community's understanding about learning-based detectors and promote research on Android security.

Therefore, in this paper, we select 12 detectors for evaluation based on the criteria such as diversity, popularity, relevance, and recency, including five String-based detectors, two Image-based detectors, and five Graph-based detectors. We develop an evaluation framework, which includes the following scenarios: code obfuscation, concept drift and AEs attack. In addition, we also evaluate the performance of detectors under the ideal setting and test the detection efficiency of various detectors. We evaluate the three different types of detectors using the unified datasets, and analyze the evaluation results to achieve the following three goals: 1) understand the capabilities of different-type detectors, 2) analyze the reasons for the difference of these detectors' performance under challenging environments, and 3) provide some suggestions from the perspectives of users and researchers. Our work aims to provide a clearer picture for learning-based Android malware detection studies. Finally, our contributions are summarized as follows:

1) We develop a systematic evaluation framework to evaluate various types of learning-based Android malware detectors under challenging environments, which fills a gap in previous research.

2) We summarize our findings and analyze the reasons for performance differences among various types of learning-based Android malware detectors under diverse challenging environments. We also provide some suggestions for both users and researchers.

3) We build three concept drift datasets and eight code obfuscation datasets, which are available on our link [1]. These datasets can be used by the research community to evaluate various detectors under challenging environments.

2 PRELIMINARY

2.1 Learning-based Android malware detector

As shown in Figure 1, the construction of learning-based detectors can be divided into four main steps: **1) Preprocessing**: This step usually starts with unpacking an APK to obtain important files such as classes.dex. Furthermore, to better understand the code in the classes.dex, the latter can be further decompiled into smali files. **2) Feature extraction**: This step extracts features from APKs, such as function calls, and permission usage. **3) Feature processing**: The features extracted in the last step are further processed to facilitate being handled by learning models, such as a vector, an image, or a graph. **4) Model training and testing**: In this step, a malware detector or multi-category classifier is trained and then tested. Once the model passes the model test, it can be deployed in realistic environments.

2.2 Code obfuscation

Code obfuscation is a technology that converts the program code into a form that is functionally equivalent but difficult to read [24]. Obfuscation techniques are originally proposed to protect the intellectual property of legitimate users, but inevitably, attackers can also use them to protect malicious code. There are many obfuscation tools, such as Obfuscapk [5], DashO [19] and Allatori [2]. Common code obfuscation techniques include renaming identifiers, inserting junk code, etc. In recent years, some new and non-trivial obfuscation techniques have been proposed, such as Reflection and Encryption. Below we give a brief description of obfuscation technologies that will be used for this evaluation.

Rebuild (RBD): disassemble and recompile the classes.dex file. **ClassRename (CR)** and **MethodRename (MR)**: rename the class

and method name to a meaningless string, respectively. **Reorder (ROR)**: change the conditional branch or insert a goto instruction to alter the structure of basic code blocks. **Reflection (REF)**: enable functions to be called by reflection mechanism and specific APIs, thus hiding the actual calling relationship. **Junk Code (JUNK)**: insert useless code that will not be executed due to arithmetic constraints. **ConstStringEncryption (CSE)**: encrypt the constant strings in code. **CallIndirection (CID)**: change the original function call relationship by indirect calls.

2.3 Concept drift

Concept drift is a phenomenon in which the statistical properties of the target variable change over time [55]. In the context of Android malware detection, researchers are primarily concerned with two practical problems related to concept drift, i.e. 1) Android software development technique evolution and 2) unseen or new malware category emergence.

Android software evolution: In practice, the samples for model training and testing are usually released at different times. Android application programming technique evolves over time due to facts such as system upgrades and technical adjustments [53]. Android software evolution incurs different code implementations of the same function, thereby altering the distribution of features (e.g., function calls). Accordingly, the classification boundary of detectors needs to be shifted to accommodate this change [55].

Unseen or new malware category emergence: In malware analysis, it is also crucial to identify the category or family to which the malicious software belongs. However, take the multi-category classification as an example, the most popular learning-based classification models are usually trained on the dataset with fixed categories, i.e., the categories to be classified are assumed to be fixed and known in advance [48]. As malware evolves and new classes emerge [36], however, newly obtained samples may not belong to any of the known categories and hence are misclassified [53].

2.4 Adversarial examples

The adversarial examples (AEs) [41] [25] [26] are generated by adding subtle perturbations to natural samples, which can cause the learning-based model to give an incorrect classification result with high confidence. The AEs generation techniques were initially studied in the field of computer vision [58]. In the field of Android malware detection, generating realistic AEs may be more challenging, since the perturbations are required to keep software functionality unchanged and can be implemented in the problem space [37]. In recent years, some tools have emerged for generating Android malware AEs [15] [60] [3], which facilitate evaluating the robustness of Android malware detectors.

3 EVALUATION FRAMEWORK DESIGN

3.1 Detectors selection

We select detectors for performance evaluation and comparison, according to the following criteria:

1) **Diversity**: Since the existing detectors fall into different categories, the selected ones should be representative and diverse.
2) **Impact**: We try to select the most popular detectors from each category as much as possible. For example, Drebin [7] has been cited for more than 2000 times, hence adopted in our evaluation.
3) **Relevance**: A detector may be the improved version of another one, which belongs to the same category. For instance, MDMC [54] is an improvement on ImgDroid [51]. Hence both MDMC and ImgDroid are selected by us. The detectors with high relevance help to thoroughly understand the effects of technological advances, making them appropriate to our evaluation.
4) **Recency**: We tend to choose the detectors proposed in the last 5 years. Accordingly, some new and promising techniques, such as the NLP-based ones [59], are covered in our work.
5) **Reproducibility**: For a fair and accurate evaluation, only the detectors that provide open-source code or key files (e.g., the files used for feature selection) are considered by us.

Accordingly, we select 12 detectors, including five String-based, two Image-based and five Graph-based, as depicted in Table 1.

3.1.1 String-based. String-based detectors extract necessary strings as features from APKs and encode them into vectors.

Drebin: Drebin [7] extracts features from the Androidmanifest.xml and classes.dex files. The extracted features are organized into 8 sets, such as hardware components and requested permissions, which are presented as strings. These feature sets are then mapped into a vector space, where each dimension is either 0 or 1. Finally, Drebin trains an SVM classifier to detect malware.

MudFlow: MudFlow [10] extracts data flows from sensitive data sources to sensitive data sinks using Flowdroid [40]. Next, MudFlow chooses an appropriate level of granularity to process these data flows. Finally, MudFlow trains a binary classifier using all samples.

RevealDroid: RevealDroid [22] extracts 44 types of features: method-level API usage, package-level API usage, reflective features and native calls. The first three features come from the classes.dex, while the last is from native binaries. These features are used to create a 1054-dimensional feature vector, and an SVM classifier is trained to detect malware.

ALDroid: ALDroid [56] extracts permission requirements, and intent action declarations from the Androidmanifest.xml file, as well as sensitive API calls from the smali files. ALDroid then selects 379 features, including 147 permissions, 126 intent actions and 106 API calls. Finally, ALDroid uses the Broad Learning System (BLS) [14] as the classifier for malware detection.

Bai's: In Bai's [12] study, 250 common features are extracted from Androidmanifest.xml and .smali files, including 50 Android permissions, 156 API calls, and 44 Intent attributes for inter-component communication. Then MLP, RF, SVM, et al. are introduced to act as the classifier.

3.1.2 Image-based. Image-based detectors use various methods to convert an APK into an image, which is then processed with a deep learning model.

ImgDroid: ImgDroid [51] obtains bytecodes from the classes.dex file, and converts bytecodes into RGB images. It trains a CNN network to distinguish between malicious and benign apps through classifying the RGB images.

MDMC: Similar to ImgDroid, MDMC [54] relies on bytecodes for malware analysis. However, MDMC doesn't directly transform bytecodes to images. Instead, it constructs the Markov images based on probability matrices of byte transfers, and then uses a CNN network to classify the Markov images.

Table 1: Overview of studied learning-based Android malware detectors.

Type	Name	Years	Feature Source	Feature Extraction	Feature processing	Classifier
String-based	Drebin [7]	2014	Androidmanifest.xml, classes.dex	Hardware components, Requested permissions, Suspicious API calls, et al.	Hash	SVM
	MudFlow [10]	2015	Androidmanifest.xml, classes.dex	Data flows of sensitive sources	Hash	SVM
	RevealDroid [22]	2018	classes.dex, native binaries	API-Usage, Reflective API, Native Call	Hash	SVM
	ALDroid [56] [27]	2020	Androidmanifest.xml, classes.dex	Used permissions, APIs and Actions	Hash	BLS
	Bai's [12] [11]	2021	Androidmanifest.xml, classes.dex	Used permissions, APIs and ICCs	Hash	SVM, RF, DT, KNN, MLP
Image-based	ImgDroid [51]	2019	classes.dex	Bytecode of classes.dex	Convert bytecode to color images	CNN
	MDMC [54]	2020	classes.dex	Bytecode of classes.dex	Convert bytecode to markov images	CNN
Graph-based	MaMa-fml [31]	2017, 2019	classes.dex	Function call relationship	Abstracte API calls to family calls	RF, KNN
	MaMa-pkg [31]	2017, 2019	classes.dex	Function call relationship	Abstracte API calls to package calls	RF, KNN
	Malscan [49]	2019	classes.dex	Function call relationship	Compute the centrality of sensitive API calls	RF, KNN
	APIGraph [59]	2020	classes.dex	Function call relationship	Upon other features	Use other classifiers
	EFCG [13]	2021	classes.dex	Function call relationship	NLP-enhanced, GCN	RF, DT, SVM, LR, et al.

3.1.3 Graph-based. Graph-based detectors extract features by constructing graph models from APKs, which are then processed using a learning-based model.

MaMa-fml: MaMa-fml is one model of MaMaDroid [31]. MaMa-fml extracts API calls from smali files and abstracts them into family calls. There are 11 families, 9 of which are from official Android documentation. The other two are self-defined and obfuscated, respectively. MaMa-fml then constructs the Markov chain on the basis of the transfer probabilities among families. Finally, the Markov chain is used as the feature vector to train a detection model.

MaMa-pkg: MaMa-pkg is another model of MaMaDroid. MaMa-pkg extracts features in the same way as MaMa-fml. The difference is that MaMa-pkg abstracts API calls into package calls. There are 366 packages from official Android documentation, and 2 packages are self-defined and obfuscated, respectively. In consequence, a 368×368 dimensional feature vector is used to represent an app.

Malscan: For each app, Malscan [49] constructs a function call graph (FCG) from its smali files. Then Malscan selects 21986 sensitive API calls based on PScout's results [8], and computes the centrality of sensitive API calls as the feature vector. Lastly, Malscan employs RF and KNN to perform classification, respectively.

APIGraph: APIGraph [59] is designed to mine semantic similarities between Android APIs. It builds a relation graph of Android APIs based on official documents. APIGraph can be applied to previous Android malware classifiers for performance improvement. Hence, we apply APIGraph to MaMa-fml and MaMa-pkg in our evaluation, denoted as APIGraph_f and APIGraph_p, respectively.

EFCG: EFCG [13] first constructs an FCG for each app, and trains a Fun2vec model to obtain the representation for every function. It then creates an enhanced FCG, and uses a Graph Convolutional Network (GCN) [46] to obtain a 100-dimensional vector representation for an enhanced FCG. Finally, EFCG employs the common classifiers such as RF and SVM to detect malware.

3.2 Research questions

We aim to answer the following research questions through experiments and analyses.

RQ1: How do various detectors perform under the ideal setting? To provide a reference for subsequent evaluations, we first evaluate the performance of various detectors on malware detection and Android software multi-category classification.

RQ2: How do different detectors perform on obfuscated samples? We evaluate the resilience of various detectors to different obfuscation techniques and analyze the factors that contribute to their differences in resilience.

RQ3: How does concept drift affect the various detectors? To evaluate the ability of different detectors to alleviate concept drift, we conduct our experiments in two scenarios, including Android software evolution and new malware category emergence.

RQ4: How do various detectors perform under AE attacks? To answer this question, we evaluate the robustness of detectors using different adversarial example generation tools.

RQ5: How efficient are various detectors? To obtain a more comprehensive understanding of various detectors, we measure their average processing time of handling a sample.

3.3 Dataset and metrics

As shown in Table 2, we employ two datasets for evaluation. Data-MD is used for Android malware detection, and Data-MC is used for Android software multi-category classification.

Data-MD is a new dataset we construct with two steps. 1) We collect a large batch of samples from Androzoo [4], spanning the years 2016-2020. 2) We upload the samples to Virustotal [44] for labeling. To ensure the quality of the datasets, we adopt some important considerations and measures. 1) Due to the concern of sample replication [61], we de-duplicate the samples in our dataset. Specifically, if two APKs have the same hash value of their classes.dex file, we consider one of them as a duplicated APK. There are 490 replicate samples in the set of collected samples. 2) Recently, some studies pointed out that certain engines in VirusTotal will dynamically flip their output labels over time. Fortunately, Zhu et al. [62] have conducted an evaluation for this problem, and provided an appropriate selection range for the Voting threshold (VT) (i.e., from 2 to 14). We adopted this suggestion, and set VT to 4. That is, if more than 4 engines label a sample as malicious, its label is set to be malicious. If all engines label the sample as benign, then its label is benign. In the end, we collect 15,356 samples. It is worth noting that none of the detectors we have selected for evaluation are from Virustotal. 3) We construct a balanced dataset since training a detector on an unbalanced dataset (i.e., the number of benign samples is much larger than that of malicious ones) will make the classification boundary biased towards benign samples. Accordingly, apps are more likely to be classified as benign, and malware detection is easier to be evaded by code obfuscation or adversarial perturbation.

Data-MC is from CICMalDroid [30][29]. This dataset has five distinct categories: Adware, Banking, SMS, Riskware, and Benign. All samples undergo initial dynamic analysis using CopperDroid (a VMI-based dynamic analysis system) [42]. Samples that fail to run due to fatal errors such as time-outs, invalid APK, and memory allocation failures are excluded from the dataset. In addition, we also de-duplicate the samples for this dataset.

We employ the commonly used evaluation metrics, including Accuracy (Acc), Precision (P), Recall (R), and F1-Score (F1), to reflect the results of malware detection and Android software multi-category classification. In addition, we also use the more advanced metric MCC [16] in RQ1 to provide a more comprehensive insight.

Table 2: The brief description of the dataset.

Android Malware Detection Dataset (Data-MD)						
Years	2016	2017	2018	2019	2020	Total
Benign	1480	1851	1890	1601	1111	7933
Malicious	1060	1514	1561	1793	1495	7423
Total	2540	3365	3451	3394	2606	15356
Android Software Multi-category Classification Dataset (Data-MC)						
Category	Adware	Banking	Riskware	SMS	Benign	Total
#Sample	1515	2506	2070	2536	4042	12669

3.4 Experimental setup

Our experiments are conducted in the following scenarios.

3.4.1 Ideal setting. We evaluate the performance of different detectors on malware detection and multi-category classification under the ideal setting in RQ1. For Data-MD and Data-MC, we randomly selected 80% of samples from each category as the training set, and the remaining 20% as the testing set. It is worth noting that RQ5 is also evaluated in this scenario.

3.4.2 Code obfuscation. We evaluate the detectors' resistance to code obfuscation on Data-MD in RQ2. We select 800 benign and 800 malicious samples from the **test set** of RQ1 as the original test set *(ORI)*. For every sample in the original test set, we impose eight different obfuscation techniques on it, respectively. We select Obfuscapk [5] as our obfuscation tool. Obfuscapk is an open-source tool used for obfuscating Android apps without accessing their source code. When obfuscating an APK, Obfuscapk first decompiles the APK to generate Smali files. Then, Obfuscapk obfuscates the smali files and repackages them to generate the obfuscated APK. Obfuscapk can implement classic obfuscation techniques and some advanced obfuscation techniques, such as Reflection. Finally, we obtain eight different **obfuscation test sets**.

It is noted that there may exist obfuscated samples in Data-MD, which negatively affect the validity of our evaluation. However, it is too challenging to detect and filter out all the obfuscated samples from Data-MD. Therefore, we manually construct a dataset without any obfuscated samples, and then use contrast experiments to decide whether the experiments over Data-MD are valuable. The results of contrast experiments indicate that the presence of obfuscated samples in Data-MD does not cause bias in the evaluation results. Hence Data-MD is suitable for our experiments on the resistance to code obfuscation. Details of the evaluation approach can be found in our link [1].

3.4.3 Concept drift. As described in Section 2.3 and RQ3, we consider two causes for concept drift, i.e., Android software evolution and unseen malware category emergence.

Android software evolution: Here we set up two scenarios to evaluate the performance of various detectors. Scenario one is designed to evaluate the detectors' resilience to aging, while Scenario two is intended to assess their ability to resist forgetting. In Scenario one, the detectors are trained on the samples of 2016 in Data-MD, and subsequently tested on the samples of 2017-2020, respectively. In Scenario two, the detectors are trained on the samples of 2020 in Data-MD, and then tested on the samples of 2016-2019, respectively. To maintain the consistency of sample quantity for training and test, a total of 2160 samples are selected from each year's dataset, with an equal number of benign and malicious samples.

New malware category emergence: To assess the effectiveness of different detectors in identifying unseen malware category samples, we select one malware category from the five categories in Data-MC as the unseen category. For example, we choose *Aware* as the unseen category. We construct the training and test sets with the remaining four categories, and then add the unseen category into the **test set**. Note that the samples from the unseen category are excluded from the training set.

3.4.4 Adversarial examples. We assess the detectors' robustness to AEs using Data-MD in RQ4. We use the training and test sets of RQ1 to train the model. We then select malware from the test set to generate AEs, which are then used to evaluate the detection models. In the following, we introduce the tools used to generate AEs. Al-Dujaili et al. [3] presented *dFGSM_k* and *rFGSM_k* to generate binary-encoded AEs of malware. *dFGSM_k* and *rFGSM_k* are both the variants of FGSM. Chen et al. [15] proposed two AEs crafting algorithms, denoted as *HIV_JSMA* and *HIV_CW*. It is worth noting that adversarial perturbations are usually imposed on features. Therefore, AE attacks take effect in the feature space. Meanwhile, constraints on the problem space also need to be considered. For example, Chen et al. [15] introduce a position constraint on feature perturbations to ensure realistic AEs can be generated in the problem space. Therefore, we also consider these constraints when employing AE generation tools to evaluate detectors.

4 EVALUATION

In this section, we show the experimental results of RQ1-RQ5. Note that all experimental setups are shown in Section 3.

4.1 RQ1: Performance under the ideal setting

Results & Analyses: Table 3 shows the Acc, P, R, F1 and MCC of various detectors on Malware detection (MD) and multi-category classification (MC) tasks in the ideal setting. On the MD task, the F1 of various detectors exceeds 0.9, and Malscan exhibits the highest performance with an F1 of 0.959. On the MC task, the performance of the detectors decreases to varying degrees, compared to that on the MD task. This can be attributed to the fact that MC is a more challenging classification task with a finer granularity. Overall, the detectors in different categories can all achieve satisfactory performance in the ideal setting.

Table 3: F1 of various detectors on MD and MC tasks.

Detector	Malware detection					Multi-category classification				
	Acc	P	R	F1	MCC	Acc	P	R	F1	MCC
Drebin	0.949	0.949	0.949	0.949	0.898	0.948	0.943	0.943	0.943	0.933
RevealDroid	0.956	0.956	0.956	0.956	0.913	0.941	0.937	0.936	0.936	0.924
MudFlow	0.940	0.940	0.940	0.940	0.880	0.908	0.907	0.907	0.907	0.880
ALDroid	0.937	0.937	0.937	0.937	0.873	0.917	0.907	0.913	0.908	0.894
Bai's	0.954	0.954	0.954	0.954	0.907	0.951	0.948	0.947	0.947	0.938
ImgDroid	0.959	0.959	0.959	0.958	0.917	0.906	0.898	0.908	0.902	0.873
MDMC	0.956	0.957	0.956	0.956	0.913	0.939	0.937	0.935	0.936	0.921
MaMa-fml	0.956	0.956	0.956	0.956	0.912	0.929	0.928	0.928	0.928	0.908
MaMa-pkg	0.956	0.957	0.956	0.956	0.913	0.940	0.934	0.938	0.935	0.923
Malscan	0.959	0.959	0.959	0.959	0.918	0.934	0.931	0.931	0.930	0.916
APIGraph_p	0.956	0.957	0.956	0.956	0.913	0.932	0.926	0.929	0.927	0.913
APIGraph_f	0.955	0.956	0.955	0.955	0.911	0.925	0.926	0.925	0.925	0.903
EFCG	0.942	0.943	0.942	0.942	0.885	0.934	0.927	0.934	0.929	0.915

F1: In the ideal setting, all three types of detectors perform well on the MD task, and Malscan shows the best performance. Moreover, the MC task is more challenging in comparison.

4.2 RQ2: Performance under code obfuscation

Results & Analyses: Figure 2 shows the performance of various detectors on the eight obfuscation test sets. The explanations of obfuscation techniques are shown in Section 2.2. The results indicate that the String-based detectors perform relatively stably, whereas the Image-based detectors experience more significant performance degradation. For example, the average F1 of ImgDroid is only 0.76. Most of the Graph-based detectors have relatively stable performance on the obfuscation test sets, but MaMa-fml and APIGraph-f are more susceptible to code obfuscation.

We further analyze how various obfuscation techniques affect our detectors. Let $F1_{ORI}$ and $F1_{RBD}$ denote the F1 score measured on the original and the obfuscation test set, respectively. Then we can use $(F1_{ORI} - F1_{RBD})/F1_{ORI} \times 100\%$ to represent the decrease ratio of F1 under code obfuscation. Table 4 shows the decreased ratio of F1 for the evaluated detectors on different obfuscation test sets. It can be seen that the Image-based detectors are significantly degraded on all the obfuscation test sets. Under the simplest obfuscation technique, *RBD*, the F1 of ImgDroid and MDMC decreases by 24.62% and 14.44%, respectively. More seriously, ImgDroid drops by 38.39% on the *CID* obfuscation test set.

The performance of the Graph-based detectors varies significantly under various obfuscation techniques. On the obfuscation test sets of *RBD*, *JUNK* and *ROR*, the F1 of detectors is essentially unchanged, while it suffers different degrees of degradation on the obfuscation test sets of *CR*, *CSE*, *CID*, and *REF*. In addition, different detectors have varying degrees of sensitivity to the same obfuscation techniques. For example, on the *CID* test set, the F1 of MaMa-fml, APIGraph-f, MaMa-pkg and Malscan decreases by 52.23%, 24.71%, 8.63% and 8.75%, respectively.

Compared to the other two types of detectors, the String-based detectors exhibit more stable performance on the obfuscation test sets. Most detectors suffer only a minimal decrease in F1 under code obfuscation. In comparison, Drebin, RevealDroid, and MudFlow are more vulnerable to some obfuscation techniques. Take Drebin for example. Its F1 decreases by 1.71% on *CR* test set, which is already the most serious performance degradation it has experienced across all the obfuscation test sets.

Table 4: The decrease ratio of F1 under obfuscation.

Tools	RBD	CR	MR	CSE	JUNK	CID	ROR	REF
Drebin	0.21%	1.71%	0.32%	0.21%	0.21%	0.11%	0.21%	0.21%
RevealDroid	0.63%	-0.73%	0.84%	0.63%	0.63%	0.95%	0.63%	0.63%
MudFlow	0.80%	0.65%	0.61%	0.80%	0.80%	0.34%	0.80%	0.80%
ALDroid	0.11%	0.00%	0.21%	0.11%	0.11%	0.00%	0.11%	0.11%
Bai's	0.42%	0.32%	0.74%	0.42%	0.42%	0.32%	0.42%	0.42%
ImgDroid	24.62%	28.22%	27.88%	29.92%	33.29%	38.39%	36.64%	25.90%
MDMC	14.44%	36.94%	13.24%	25.97%	14.17%	24.05%	14.84%	19.19%
MaMa-fml	0.55%	17.27%	0.65%	17.04%	0.55%	52.23%	0.55%	13.92%
MaMa-pkg	0.53%	2.58%	0.95%	2.14%	0.53%	5.30%	0.53%	2.47%
Malscan	0.53%	1.70%	1.38%	1.81%	0.53%	8.63%	0.53%	1.81%
APIGraph_p	0.53%	3.24%	0.84%	2.35%	0.53%	8.75%	0.53%	1.81%
APIGraph_f	0.65%	15.36%	0.93%	10.23%	0.65%	24.71%	0.65%	3.66%
EFCG	0.85%	2.05%	1.29%	2.50%	0.85%	3.74%	0.85%	1.61%

F2: The String-based detectors are relatively stable under code obfuscation. The Image-based detectors deteriorate more severely (The F1 is reduced by 25.48% on average). The Graph-based detectors perform well under most obfuscation techniques, but their performance significantly decreases when facing some obfuscation techniques such as *CID* and *CR*.

Reasons & Explanations: Here we further study why the three types of detectors have different resilience to various obfuscation technologies.

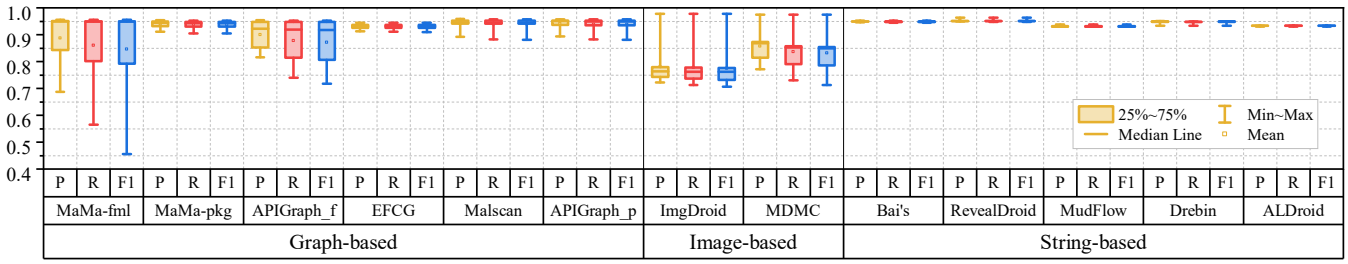


Figure 2: P, R, F1 of detectors on the original and obfuscation test sets.

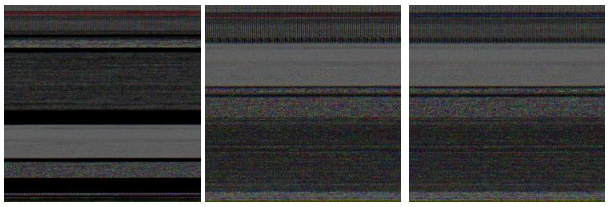


Figure 3: Impact of code obfuscation on color images.

1) There are two reasons for the relatively strong resilience of the String-based detectors to code obfuscation. First, as shown in Table 1, the String-like features used by these detectors usually come from classes.dex, AndroidManifest.xml and native binaries, while most of obfuscation techniques only modify classes.dex. Second, the obfuscation techniques usually alter a fraction of the string-like features with a small impact. For illustration, we suppose a String-based detector uses API calls as its features. Under the *CR* obfuscation (i.e., changing functions' name), only some of the pre-existing API calls vanished, changing the corresponding elements in the string-like feature from 1 to 0 and keeping the remaining elements unchanged.

2) In general, the Image-based detectors utilize all the information contained in classes.dex for classification. However, the latter can be easily changed by obfuscation. For illustration, we extract the feature used by ImgDroid from an APK file. We then use *RBD* and *CID* to modify the APK file, respectively. The original features and the changed ones are all represented by a color image, as depicted in Figure 3. Clearly, the changed features are significantly different from the original ones. Therefore, it is hard for ImgDroid to resist even simple obfuscation techniques like *RBD*. On the other hand, the visual distinction between *CID* and *RBD* Bytecode color images is less pronounced. This alignment effect is not by coincidence. This is because *RBD* acts as the last step of all obfuscation techniques, dominating the visual pattern of Bytecode feature images. This also explains why ImgDroid suffers similar performance degradation on the other obfuscation test sets.

3) Graph-based detectors mainly depend on function call graphs (FCGs) and their variants for classification. Table 5 shows the change in the number of nodes and edges in FCGs before and after code obfuscation. *Edge_diff* (*Node_diff*) denotes the discrepancy in the edge (node) count between the original and obfuscated app. Furthermore, to assess the statistical difference of FCGs before and after obfuscation, we employ the Kolmogorov-Smirnov test [33] to evaluate the distribution of node and edge count before and after obfuscation. *Node_p* and *Edge_p* denote the *p*-value of test result. A *p*-value less than 0.05 indicates that the distribution of node (or edge) count for the obfuscated app significantly differs from that for the original app. The results demonstrate that the *CID* obfuscation substantially increases the node and the edge count by 27.728% and 19.473%, respectively. Consequently, most Graph-based detectors exhibit subpar performance on the *CID* obfuscation test set. In particular, MaMa-fml suffers a more severe performance decline due to its coarse-grained features. Obfuscation techniques, such as *CR*, *MR*, *CSE*, and *REF*, exert a marked effect on FCGs, with diverse node distribution and edge count compared to the original apps. Conversely, the three obfuscation strategies, namely *RBD*, *JUNK* and *ROR*, exert a relatively low impact on FCGs. As a result, the Graph-based detectors experience slight performance degradation on these obfuscation test sets.

Table 5: Impact of code obfuscation on function call graphs.

Obf.Tec	Node_diff	Node_p	Edge_diff	Edge_p
RBD	0.000%	0.317	0.000%	0.317
CR	+0.155%	*<0.05*	+0.083%	*<0.05*
MR	+0.131%	*<0.05*	0.000%	0.317
CSE	+0.140%	*<0.05*	+0.676%	*<0.05*
JUNK	0.000%	0.317	0.000%	0.317
ROR	0.000%	0.317	0.000%	0.317
CID	+27.728%	*<0.05*	+19.473%	*<0.05*
REF	+0.044%	*<0.05*	-0.174%	*<0.05*

F3: The obfuscation-resilience of detectors is closely related to the selected features. Different features exhibit distinct levels of robustness against various obfuscation techniques. Moreover, utilizing global information extracted from classes.dex files as features can increase the risk of detection performance degradation caused by obfuscation.

4.3 RQ3: Performance under concept drift

(1) Android software evolution

Results & Analyses: We assess various detectors under two scenarios. In *Scenario one*, the training samples are all from 2016. In *Scenario two*, the training samples are all from 2020. Figure 4 shows the F1 scores of various detectors under two scenarios to reflect their resilience to concept drift resulted by Android software development technique evolution.

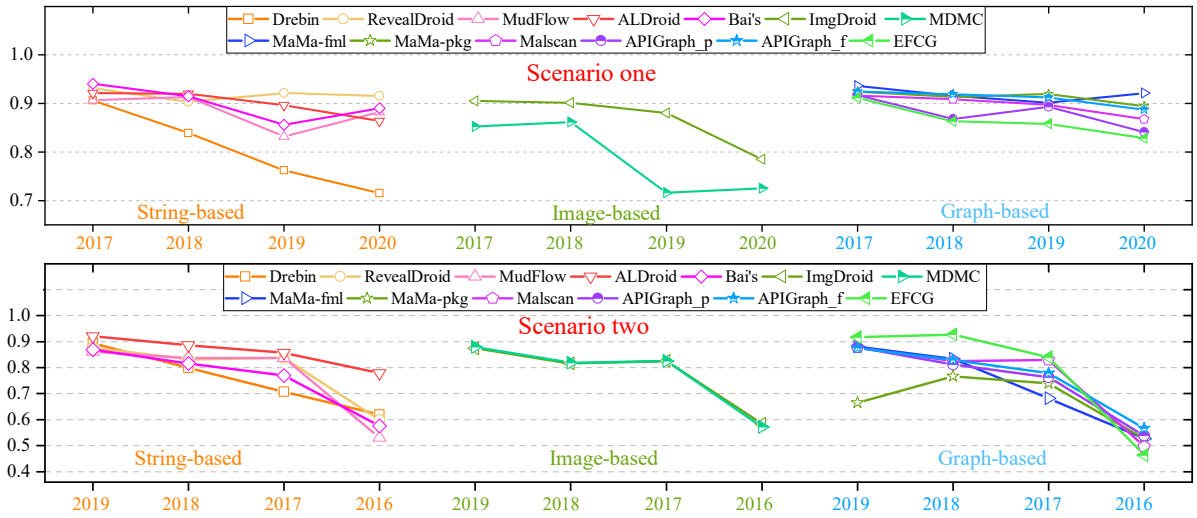


Figure 4: The F1 of various detectors under scenario one and scenario two.

1) In Scenario one, the String-based and the Image-based detectors suffer a more significant decline in F1 than the Graph-based detectors. Moreover, for most of the detectors, a larger time gap between the training and test sets results in a sharper decline in F1. For instance, Drebin's F1 drops from 0.901 on the 2017 test set to 0.839, 0.762, and 0.715 on the 2018, 2019, and 2020 test sets, respectively. 2) In Scenario two, the F1 scores of various detectors decrease more rapidly than in Scenario one. This is because the models trained with new samples may pay more attention to the features associated with new development techniques, making them perform worse on old samples. Comparatively, the Graph-based detectors suffer more severe performance degradation in Scenario two. For example, EFCG's F1 drops from 0.912 on the 2019 test set to 0.463 on the 2016 test set. Similarly, Malscan's F1 decreases from 0.876 on the 2019 test set to 0.497 on the 2016 test set. Furthermore, the results indicate that coarse-grained features are more resilient than fine-grained features. For instance, in both scenarios, MaMa-fml exhibits a slower decline in F1 compared to MaMa-pkg.

F4: When concept drift occurs, all three types of detectors experience varying degrees of performance degradation. Overall, Image-based detectors show the most significant performance drop under concept drift. In addition, using a new training set leads to more severe performance degradation for detectors than using an old training set.

Reasons & Explanations: Android software development technique undergoes continuous evolution, changing the features extracted from apps. We will explore which features undergo the most significant changes over time, resulting in the degradation of detector performance. First, we randomly select 2000 samples with comparable sizes from each year between 2017 and 2020. We identify the top-100 commonly-used sensitive API calls, top-40 commonly-used permissions and actions in the samples of 2016. We then analyze their frequency of use in the samples from 2017 to 2020. As shown in Figure 5, the frequency of API usage decreases each year. Its median is 30.04% in 2017, which decreases to 28.24%

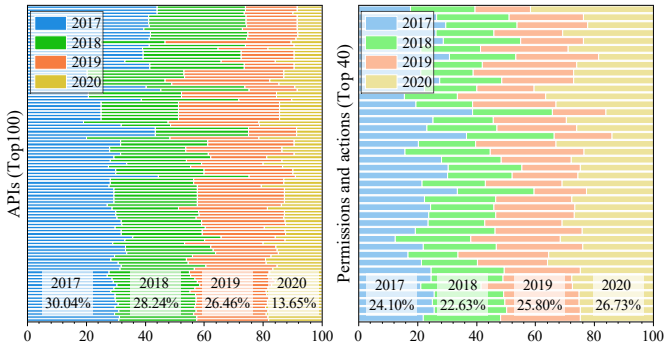


Figure 5: The usage frequency of APIs, permission and actions in different years.

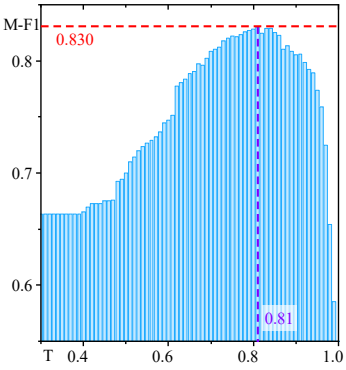
and 26.46% in 2018 and 2019, respectively. In 2020, its median further decreases to 13.65%. In contrast, the frequency of permissions and actions remains more stable, with the median hovering around 25% across the four different years.

The above analyses indicate that using too many sensitive APIs as features may reduce a model's robustness against concept drift. As shown in Figure 4, ALDroid selects more permissions and actions as its features, while Malscan and Drebin choose more sensitive APIs as their features. Therefore, ALDroid has stronger resilience compared to the other two detectors. Although APIGraph helps to mitigate semantic differences between APIs from different years, its effectiveness in countering software evolution is actually limited, particularly over significant time spans. Moreover, through comparing MaMa-fml and MaMa-pkg, we observe that coarse-grained features exhibit greater resilience to software evolution, as an API's name may change but its family name often remains unchanged.

F5: The negative impact of concept drift caused by software evolution has a stronger association with the sensitive APIs adopted in features. This effect is even more significant when the time span between training and test samples is large.

(2) New malware category emergence

To explore the potential of existing detectors in identifying new-category samples, now we focus on the concept drift induced by new malware category emergence. In our experiments, the test set contains the samples of a new category unseen in the training set. We sequentially evaluate all the detectors under 100 thresholds ranging from 0.01 to 1. We choose the threshold that yields the highest macro average F1-score (M-F1) as the final threshold. For illustration, we consider EFCG as an example and show its M-F1 under all thresholds in the left part of Figure 6. Clearly, EFCG achieves the highest M-F1 0.83 when the threshold is 0.81. We then show the final threshold and the highest M-F1 of every detector in the right part of Figure 6. It can be seen that the Graph-based detectors exhibit the strongest capability for recognizing new-category samples, with each detector achieving an M-F1 above 0.8. This is because the Graph-based detectors use the features that can better describe software behavior and possess stronger discriminability. Moreover, RevealDroid, Drebin and Bai's in the category of String-based detectors also have distinct advantages in identifying samples from new malware categories. Furthermore, no detector achieves an F1 above 0.9. When the distribution of test samples differs from that of training samples, the decision boundary learned through training should be shifted to adapt to the new distribution. This tells us that detectors should be continuously updated to handle new malware category emergence.



Detector	T	F1
Drebin	0.59	0.871
RevealDroid	0.83	0.875
MudFlow	0.84	0.625
ALDroid	0.33	0.775
Bai's	0.75	0.868
ImgDroid	0.99	0.762
MDMC	0.99	0.854
MaMa-fml	0.58	0.827
MaMa-pkg	0.83	0.843
Malscan	0.83	0.870
APIGraph_p	0.83	0.875
APIGraph_f	0.79	0.843
EFCG	0.81	0.830

Figure 6: Threshold (T) and F1 in recognizing new category.

F6: All types of detectors have limited ability to recognize the samples of unseen categories. Comparatively speaking, Graph-based and String-based detectors perform better than Image-based detectors when new malware categories emerge.

4.4 RQ4: Performance under adversarial attacks

We utilize *Evasive Detection Ratio (EDR)* to assess the detectors' capability to defend against AEs. *EDR* is defined as the ratio of the number of evasive malware ($N_{evasion}$) to the total number of malware (N_{total}), i.e., $EDR = N_{evasion}/N_{total}$. Figure 7 illustrates the defense capability of various detectors in resisting AE attacks.[1] Obviously, the Graph-based detectors (e.g., MaMa-pkg and MaMa-fml) are usually more robust than the String-based detectors (e.g., Drebin and Bai's). Our explanations are given below. The existence of AEs is due to the imperfect decision boundary learned through model training [37]. Note that the String-like features are not good at accurately describing the runtime behavior of malware. Accordingly, the classification boundaries learned by the String-based detectors are not good enough. Contrarily, the Graph-based features provide a more accurate description for malware behavior, making the learned classification boundary closer to the ideal one. Under this situation, it is very difficult for attackers to launch effective AE attacks. Furthermore, MaMa-pkg performs more robustly than MaMa-fml in resisting AEs. This is because MaMa-fml uses coarser-granularity features obtained through clustering different package-level functions into a family-level function. Accordingly, these features describe malware behavior more vaguely, leaving more room for AEs.

[1]Due to space limit, we only show the representative of the experimental results.

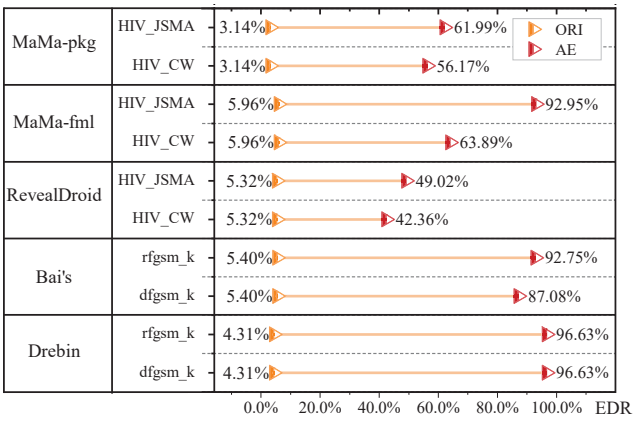


Figure 7: *EDR* of detectors on natural samples and AEs.

Moreover, RevealDroid demonstrates stronger robustness compared to other detectors. This can be attributed to the fact that RevealDroid utilizes 454 native calls as features, while the AE attacks (i.e., *HIV_JSMA* and *HIV_CW*) cannot change native binaries. In addition, we do not evaluate the defense of the Image-based detectors against AEs. In fact, it is not hard to find adversarial perturbations for a bytecode image to induce misclassification. However, it is very challenging or even impossible to implement these perturbations at the APK level [37][18]. That is, the feature-space perturbations usually cannot be appropriately mapped into the problem-space modification of the Android app's source code.

F7: Graph-based detectors are usually more robust than String-based under AE attacks. Furthermore, Image-based detectors are less exposed to AE attacks, since generating realistic AEs based on a perturbed bytecode image is too challenging to achieve.

4.5 RQ5: Efficiency

To evaluate the efficiency of each detector, we calculate the size of feature encoding model and classification model, as well as the average test time of each sample in Data-MD.

As shown in Table 6, only Drebin, APIGraph, and EFCG require a feature encoding model. ImgDroid, MDMC and EFCG need a

Table 6: Efficiency comparison for various detectors.

Detector	Feature model	Classification model	Testing time
Drebin	13870 KB	1169 KB	88.35s
RevealDroid	~	556 KB	125.21s
MudFlow	~	707 KB	55.43s
ALDroid	~	1393 KB	2.86s
Bai's	~	626 KB	2.54s
ImgDroid	~	29284 KB	1.52s
MDMC	~	29282 KB	1.83s
MaMa-fml	~	322 KB	107.35s
MaMa-pkg	~	630 KB	110.42s
Malscan	~	372 KB	3.65s
APIGraph_p	3795 KB	554 KB	118.76s
APIGraph_f	3795 KB	556 KB	109.64s
EFCG	1.05 GB	1261 KB	6.84s

larger classification model. The Image-based detectors ImgDroid and MDMC are the most efficient, whose average test time is only 1.52s and 1.83s, respectively. This is because that the Image-based detectors do not require decompiling the classes.dex file, which is a time-consuming task. The String-based detectors are generally less efficient due to their need for parsing multiple files. RevealDroid is particularly inefficient because it requires extra time for extracting reflective features and native calls. The Graph-based detectors often take a long time to build features, because decompilation and graph construction require additional time overhead.

F8: Image-based detectors have the highest detection efficiency, but their classification models require more storage space. Graph-based detectors are usually the most inefficient in malware detection.

5 LESSONS

Based on the above experiments and analyses, we comprehensively evaluate the studied detectors from different aspects in Figure 8. We summarize the lessons learned for Android malware detector users and researchers below.

5.1 Lessons for users

Our suggestions for users are summarized below. 1) For malware detection, we recommend using RevealDroid, MaMa-pkg, Bai's and Malscan, as they are more robust under challenging environments. For multi-category classification, we recommend using Drebin, RevealDroid, Malscan and APIGraph_p, because they have relatively stronger capabilities in recognizing unseen malware category samples. 2) If a user aims to analyze the malicious behavior of malware, we recommend utilizing Graph-based detectors, as they can provide a more comprehensive description of an app's runtime behavior. Additionally, String-based detectors (e.g. MudFlow) may also help to achieve this goal since they can utilize interpretable techniques to identify the key features that facilitate a better understanding of malicious behaviors. 3) For the task of large-scale malware detection, we recommend using the Image-based detectors as the first defense line, because their processing time is short, less than 2s for a sample on average. To further prevent malware from evading detection, these detectors can be used in collaboration with other tools. For example, we can use MDMC for detection first, and then use the other more robust tools (e.g. Malscan) to recheck the software identified

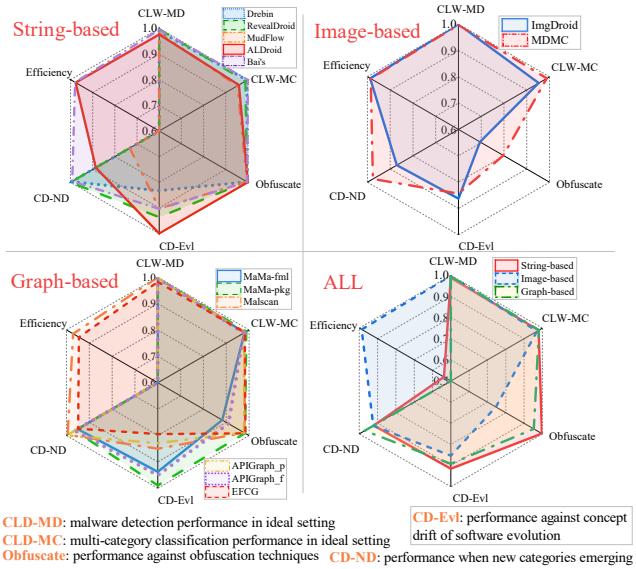


Figure 8: Comprehensive evaluation of various detectors.

as benign. 4) For the task of on-device detection, we recommend using Malscan and Bai's, as their classification models are more lightweight, do not require feature embedding, and have a certain guarantee of detection speed. In particular, we do not recommend the EFCG detector because its feature-embedding model occupies a large storage space. 5) Last but not least, although the current detectors have achieved good performance under the ideal setting, they perform unstably under challenging environments. Therefore, we suggest combining the learning-based detectors with manual analysis techniques in malware detection. It is also worth exploring how to make the detectors collaborate more harmoniously with manual analyses.

5.2 Lessons for researchers

The following suggestions may help researchers to build a more resilient detector under challenging environments.

How to resist code obfuscation? **(A)** Selecting the feature source that is less considered by most obfuscation tools. For example, more than 95% of obfuscation techniques target classes.dex. So extracting features from other files can help to enhance the robustness against obfuscation. **(B)** Selecting the features that are stable when facing obfuscation. For instance, extracting features directly based on class names or function names is not recommended, since the latter can be easily changed by obfuscation. Instead, we should give the features richer semantic information. For the Graph-based detectors, we can assign every node in the feature with node attributes. For the Image-based detectors, we can highlight those meaningful bytecodes instead of treating all bytecodes equally. **(C)** Constructing multi-view features. Features possess different defense capabilities for code obfuscation. It is difficult to count on one single feature to defend against all obfuscation techniques [21]. Therefore, combining multiple features that complement each other in resisting obfuscation is a better choice.

How to alleviate concept drift? **(A)** Solely relying on features to withstand concept drift is not advisable. In fact, concept drift

significantly impacts all three types of detectors, particularly when the samples used for model training are too old. **(B)** A more effective countermeasure might be to introduce a concept drift detection and adaptation mechanism [55]. For example, model updates should be listed on the agenda, once the detection performance is observed to be rapidly decreasing (even if it is still satisfactory). **(C)** Since early malware are likely still prevalent, model updates should not only take into account newly emerged samples. The updated model still needs to maintain the recognition ability for the old samples. Incremental learning [32] can well balance model stability and model plasticity, and hence is suitable for updating the detectors.

How to defend against AEs? Apart from traditional defense methods such as adversarial training, increasing the difficulty of launching attacks in feature space and problem space is the key to resisting AEs. **(A)** In feature space, enhancing the representative capability of features usually helps to defend against AEs. Graph-based features are preferred over String-based features in terms of AEs defense, and fine-grained graph-based features bring about stronger robustness than coarse-grained ones. In addition, sophisticated feature transformation can also be used to augment the robustness of features against AEs [27]. **(B)** In problem space, selecting feature sources that are difficult to modify can cause trouble for AE attackers. For example, modifying native binaries requires advanced assembly language skills, which is much more difficult than modifying the classes.dex file. On the other hand, blocking the transition from feature space attacks to problem space attacks is also important for AEs defense. For example, it is extremely hard to accurately translate the pixel-level perturbations on bytecode images into code modifications on Android apps, due to the absence of precise meanings associated with pixel values, making it difficult to generate AEs in problem space.

6 THREAT TO VALIDITY

Here we discuss the threats to validity of this study.

1) Reproduction: When reproducing detectors, we refer to the study of Daniel et al. [6] to avoid pitfalls. For example, we strictly require that test data or other background information that is not usually available cannot affect the training process, in order to avoid data snooping. **2) Dataset:** Due to the limit in time and resources, we do not start from scratch when preparing the data. Instead, we download data from publicly available datasets. We then conduct de-duplicating, re-labeling and balancing on the data, in order to improve their quality and make them better serve our evaluation. Note that the label aggregation strategy may lose some interesting true malware. In the future, we will use more advanced strategies to label samples if they are proposed. **3) Obfuscation tool:** We select Obfuscapk [5] because it is open-source and supports many advanced obfuscation techniques in addition to those trivial ones. Furthermore, Obfuscapk is a new obfuscation tool. The obfuscated samples generated by Obfuscapk have a lower likelihood of being included in the original dataset. Moreover, we use a commercial obfuscation tool Allatori [2] and show its experimental results in our link [1]. It can be seen the impact of various obfuscation tools on different detector types demonstrates similar patterns. **4) AE attack:** Up to now, almost all the AE generation algorithms are delicately developed for a pre-determined detector. Their AEs are difficult to fool different detectors. Hence we cannot evaluate the detectors under the same AE attacks. Instead, we produce different AEs for these detectors, and analyze the latter's robustness against their own AEs.

7 RELATED WORK

Here we summarize the existing studies and highlight the differences between our work and them.

Literature review: Qiu et al. [38] surveyed the Android malware detection with deep neural networks. Wu et al. [47] presented a review of machine learning based Android malware static detection technology. Tam et al. [43] reviewed the existing Android malware analysis techniques and analyzed their performance against evolving malware. Yan et al. [52] summarized the attacks and defenses for the learning-based malware detectors of Windows PE, Android and PDF, etc.

Empirical study: Daoudi et al. [17] explored the reproducibility of 5 machine learning-based Android malware detectors. TESSERACT [36] proposed a set of constraints to mitigate spatial and temporal biases which can inflate the performance of detectors, and evaluated three detectors in the corresponding setting. Borja et al. [34] constructed a unified evaluation framework and evaluated 10 detectors. Sawadogo et al. [39] investigated the impact of data imbalance on detectors. Ma et al. [28] provided an empirical study of learning-based PE malware family classification methods. Zhan et al. [57] conducted an empirical study of Android's third-party library detection for Android apps. Alejandro et al. [23] studied the dynamic features of Android apps released from 2011 to 2018.

In summary, the main differences between our study and existing studies include the scale and settings of experiments. 1) We considered three types of Android malware detectors, and selected a total of 12 detectors for performance evaluation. 2) We evaluated the selected detectors under three challenging environments, i.e., code obfuscation, concept drift, and AEs. These settings are realistic and will be encountered in the real world. Through extensive experiments, we aim to understand how these challenging factors impact different types of detectors, reveal the reasons behind the strengths and weaknesses of various detectors, and provide practical suggestions from the perspectives of both users and developers.

8 CONCLUSION

Recently, there has been growing concern over the performance of learning-based Android malware detectors under challenging environments. In this paper, we classify the existing detectors into three categories (i.e., String-based, Image-based and Graph-based), and conduct a comprehensive evaluation for them in the scenarios of code obfuscation, concept drift and adversarial attack. Through extensive experiments and deep analyses, we evaluate these detectors' performance and figure out the reasons behind their diverse performance. We also sum up the lessons and offer practical suggestions for both detector users and researchers. We hope our work can provide valuable assistance to the security community.

ACKNOWLEDGMENTS

This work is supported in part by the Fundamental Research Funds for the Central Universities, HUST: 2022JYCXJJ035.

REFERENCES

[1] 2023. https://github.com/Maruko912/AMDs.

[2] 2023. Allatori. https://allatori.com/.

[3] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. 2018. Adversarial deep learning for robust detection of binary encoded malware. In *Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 76–82.

[4] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)* (Austin, Texas). New York, NY, USA, 468–471.

[5] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* 11 (2020), 100403.

[6] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *Proceedings of the USENIX Security Symposium*.

[7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*, Vol. 14. 23–26.

[8] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 217–228.

[9] AV-TEST. 2022. Total amount of malware and PUA under Android. https://portal.av-atlas.org/malware/statistics.

[10] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, Vol. 1. 426–436.

[11] Yude Bai, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Duoyuan Ma. 2020. Unsuccessful Story about Few Shot Malware Family Classification and Siamese Network to the Rescue. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE) (ICSE '20)*. 1560–1571.

[12] Yude Bai, Zhenchang Xing, Duoyuan Ma, Xiaohong Li, and Zhiyong Feng. 2021. Comparative analysis of feature representations and machine learning methods in android family classification. *Computer Networks* 184 (2021), 107639.

[13] Minghui Cai, Yuan Jiang, Cuiying Gao, Heng Li, and Wei Yuan. 2021. Learning features from enhanced function call graphs for Android malware detection. *Neurocomputing* 423 (2021), 301–307.

[14] CL Philip Chen and Zhulin Liu. 2017. Broad learning system: An effective and efficient incremental learning system without the need for deep architecture. *IEEE transactions on neural networks and learning systems* 29, 1 (2017), 10–24.

[15] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. 2019. Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security* 15 (2019), 987–1001.

[16] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics* 21 (01 2020).

[17] Nadia Daoudi, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2021. Lessons learnt on reproducibility in machine learning based Android malware detection. *Empirical Software Engineering* 26, 4 (2021), 1–53.

[18] Asim Darwaish, Farid Naït-Abdesselam, Chafiq Titouna, and Sumera Sattar. 2021. Robustness of Image-based Android Malware Detection Under Adversarial Attacks. In *Proceedings of the 2021 IEEE International Conference on Communications (ICC)*. 1–6.

[19] DashO. 2022. DashO. https://www.preemptive.com/products/dasho/overview.

[20] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. 2018. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Transactions on Information Forensics and Security* 13, 8 (2018), 1890–1905.

[21] Cuiying Gao, Minghui Cai, Shuijun Yin, Gaozhun Huang, Heng Li, Wei Yuan, and Xiapu Luo. 2023. Obfuscation-Resilient Android Malware Analysis Based on Complementary Features. *IEEE Transactions on Information Forensics and Security* 18 (2023), 5056–5068.

[22] J. Garcia, M. Hammad, and Sam Malek. 2017. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. In *Transactions on Software Engineering and Methodology (TOSEM)*.

[23] Alejandro Guerra-Manzanares, Marcin Luckner, and Hayretdin Bahsi. 2022. Concept Drift and Cross-Device Behavior: Challenges and Implications for Effective Android Malware Detection. *Computers & Security* (2022), 102757.

[24] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 421–431.

[25] Heng Li, Zhang Cheng, Bang Wu, Liheng Yuan, Gao Cuiying, Wei Yuan, and Xiapu Luo. 2023. Black-box Adversarial Example Attack towards FCG Based Android Malware Detection under Incomplete Feature Information. In *Proceedings of the USENIX Security Symposium*.

[26] Heng Li, ShiYao Zhou, Wei Yuan, Jiahuan Li, and Henry Leung. 2020. Adversarial-Example Attacks Toward Android Malware Detection System. *IEEE Systems Journal* 14, 1 (2020), 653–656.

[27] Heng Li, Shiyao Zhou, Wei Yuan, Xiapu Luo, Cuiying Gao, and Shuiyan Chen. 2021. Robust android malware detection against adversarial example attacks. In *Proceedings of the Web Conference 2021*. 3603–3612.

[28] Yixuan Ma, Shuang Liu, Jiajun Jiang, Guanhong Chen, and Keqiu Li. 2021. A comprehensive study on learning-based PE malware family classification methods. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1314–1325.

[29] Samaneh Mahdavifar, Dima Alhadidi, Ali Ghorbani, et al. 2022. Effective and efficient hybrid android malware classification using pseudo-label stacked auto-encoder. *J. Netw. Syst. Manag.* 30, 1 (2022), 1–34.

[30] Samaneh Mahdavifar, Andi Fitriah Abdul Kadir, and et al. 2020. Dynamic android malware category classification using semi-supervised deep learning. In *Proceedings of DASC/PiCom/CBDCom/CyberSciTech*.

[31] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*.

[32] Marc Masana, Xialei Liu, Bartłomiej Twardowski, Mikel Menta, Andrew D. Bagdanov, and Joost van de Weijer. 2022. Class-Incremental Learning: Survey and Performance Evaluation on Image Classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022), 1–20.

[33] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

[34] Borja Molina-Coronado, Usue Mori, Alexander Mendiburu, and Jose Miguel-Alonso. 2023. Towards a fair comparison and realistic evaluation framework of android malware detectors based on static analysis and machine learning. *Computers & Security* 124 (2023), 102996.

[35] Ali Muzaffar, Hani Ragab Hassen, Michael A. Lones, and Hind Zantout. 2022. An in-depth review of machine learning based Android malware detection. *Computers & Security* 121 (2022), 102833.

[36] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA, 729–746.

[37] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P)*. 1332–1349.

[38] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. 2020. A survey of android malware detection with deep neural models. *ACM Computing Surveys (CSUR)* 53, 6 (2020), 1–36.

[39] Zakaria Sawadogo, Gervais Mendy, Jean Marie Dembele, and Samuel Ouya. 2022. Android malware detection: Investigating the impact of imbalanced data-sets on the performance of machine learning models.. In *Proceedings of the 24th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 435–441.

[40] Siegfried Rasthofer Steven Arzt and et al. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI))*. Association for Computing Machinery.

[41] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *Proceedings of the 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

[42] Kimberly Tam, Aristide Fattori, Salahuddin Khan, and Lorenzo Cavallaro. 2015. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the NDSS Symposium 2015*. 1–15.

[43] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 1–41.

[44] Virustotal. 2022. Virustotal. https://www.virustotal.com/.

[45] Sinan Wang, Yibo Wang, Xian Zhan, Ying Wang, Yepang Liu, Xiapu Luo, and Shing-Chi Cheung. 2022. APER: Evolution-Aware Runtime Permission Misuse Detection for Android Apps. In *Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 125–137.

[46] Max Welling and Thomas N Kipf. 2016. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 2017 International Conference on Learning Representations (ICLR)*.

[47] Qing Wu, Xueling Zhu, and Bo Liu. 2021. A survey of android malware static detection technology based on machine learning. *Mobile Information Systems* 2021 (2021).

[48] Yueming Wu, Shihan Dou, Deqing Zou, Wei Yang, Weizhong Qiang, and Hai Jin. 2022. Contrastive Learning for Robust Android Malware Familial Classification. *IEEE Transactions on Dependable and Secure Computing* (2022).

[49] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. 2019. MalScan: Fast Market-Wide Mobile Malware Scanning by Social-Network Centrality Analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 139–150.

[50] Yueming Wu, Deqing Zou, Wei Yang, Xiang Li, and Hai Jin. 2021. Homdroid: detecting android covert malware by social-network homophily analysis. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis (ISSTA)*. 216–229.

[51] Xusheng Xiao and Shao Yang. 2019. An image-inspired and cnn-based android malware detection approach. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1259–1261.

[52] Senming Yan, Jing Ren, Wei Wang, Limin Sun, Wei Zhang, and Quan Yu. 2023. A Survey of Adversarial Attack and Defense Methods for Malware Classification in Cyber Security. *IEEE Communications Surveys & Tutorials* 25, 1 (2023), 467–496.

[53] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, and G. Wang. 2021. CADE: Detecting and explaining concept drift samples for security applications. In *Proceedings of the USENIX Security Symposium*.

[54] Baoguo Yuan, Junfeng Wang, Dong Liu, Wen Guo, Peng Wu, and Xuhua Bao. 2020. Byte-level malware classification based on markov images and deep learning. *Computers & Security* 92 (2020), 101740.

[55] Liheng Yuan, Heng Li, Beihao Xia, Cuiying Gao, Mingyue Liu, Wei Yuan, and Xinge You. 2022. Recent Advances in Concept Drift Adaptation Methods for Deep Learning. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*. 5654–5661. Survey Track.

[56] Wei Yuan, Yuan Jiang, Heng Li, and Minghui Cai. 2019. A Lightweight On-Device Detection Method for Android Malware. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2019).

[57] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. 2020. Automated third-party library detection for android applications: Are we there yet?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 919–930.

[58] Jiliang Zhang and Chen Li. 2020. Adversarial Examples: Opportunities and Challenges. *IEEE Transactions on Neural Networks and Learning Systems* 31, 7 (2020), 2578–2593.

[59] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, and Min Yang. 2020. Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[60] Kaifa Zhao, Hao Zhou, Yulin Zhu, Xian Zhan, Kai Zhou, Jianfeng Li, Le Yu, Wei Yuan, and Xiapu Luo. 2021. Structural Attack against Graph Based Android Malware Detection. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. New York, NY, USA, 3218–3235.

[61] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F Bissyandé, Jacques Klein, and John Grundy. 2021. On the impact of sample duplication in machine-learning-based android malware detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–38.

[62] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. 2020. Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. 2361–2378.

[63] Deqing Zou, Yueming Wu, Siru Yang, Anki Chauhan, Wei Yang, Jiangying Zhong, Shihan Dou, and Hai Jin. 2021. IntDroid: Android malware detection based on API intimacy analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–32.