12-2021

Empirical evaluation of minority oversampling techniques in the context of Android malware detection

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Nguyen Binh Duong TA
Singapore Management University, donta@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Empirical Evaluation of Minority Oversampling Techniques in the Context of Android Malware Detection

Lwin Khin Shar[§], Ta Nguyen Binh Duong[§], David Lo
School of Computing and Information Systems
Singapore Management University
Email:{lkshar, donta, davidlo}@smu.edu.sg

***Abstract*—In Android malware classification, the distribution of training data among classes is often imbalanced. This causes the learning algorithm to bias towards the dominant classes, resulting in mis-classification of minority classes. One effective way to improve the performance of classifiers is the synthetic generation of minority instances. One pioneer technique in this area is Synthetic Minority Oversampling Technique (SMOTE) and since its publication in 2002, several variants of SMOTE have been proposed and evaluated on various imbalanced datasets. However, these techniques have not been evaluated in the context of Android malware detection. Studies have shown that the performance of SMOTE and its variants can vary across different application domains. In this paper, we conduct a large scale empirical evaluation of SMOTE and its variants on six different datasets that reflect six types of features commonly used in Android malware detection. The datasets are extracted from a benchmark of 4,572 benign apps and 2,399 malicious Android apps, used in our previous study. Through extensive experiments, we set a new baseline in the field of Android malware detection, and provide guidance to practitioners on the application of different SMOTE variants to Android malware detection.**

***Keywords*-malware detection; oversampling; imbalanced learning; SMOTE; SMOTE variants; Android malware**

I. Introduction

Android is the most popular consumer operating system in the world - it runs in more than two billions of devices. Due to this popularity and its open-source nature, attacks on Android has been rising significantly. For instance, Symantec [1] reported that in 2018, it detected an average of 10,573 mobile malware per day; found that one in 36 mobile devices had high risk apps installed; and one in 14.5 apps accesses high risk user data. Hence, classification of Android malware in the wild is currently an active area of research.

Many research approaches have built Android malware detection models based on sequence of API calls [2]–[4], use of API calls [5]–[8] or frequency of API calls [9], [10]. To extract these features, in general two types of techniques are used — static analysis [4], [6], [8], [10]–[12] and dynamic analysis [2], [13].

Regardless of type of features and feature extraction techniques used, the ratio of training data (benign-to-malware) is often imbalanced. For example, datasets used in Android malware classification approaches [3], [4], [8], [11], [12], [14], [15] have various imbalanced ratios of 22, 12, 6, 2.1, 1.15, 0.35, 0.24, respectively. This causes the learning algorithm to bias towards the dominant classes, resulting in misclassification of minority classes. One effective way to improve the performance of classifiers is the synthetic generation of minority instances. Synthetic Minority Oversampling Technique (SMOTE) [16] is one of the pioneer techniques and since its publication in 2002, several variants of SMOTE have been proposed [17]. However, most of the malware classification approaches including the above-mentioned ones have not used oversampling techniques to address data imbalanced data. Although some studies [17] have evaluated these oversampling techniques in the context of defect/bug prediction, they have not been evaluated in the context of Android malware classification. Studies have shown that the use of oversampling can improve defect prediction performance but their performances may vary across different application domains [18]. This necessitates a thorough evaluation of SMOTE and its variants to facilitate SMOTE applications to Android malware classification.

In this paper, we conduct a large scale empirical evaluation of SMOTE and its variants on six different datasets containing 4,572 benign apps and 2,399 malicious apps. For training Android malware classifiers, typically three types of features — sequence of API calls (e.g. [2]–[4]), use of API calls (e.g. [5]–[8]), and frequency of API calls (e.g. [9], [10]) — are used. And these features can be extracted using static analysis or dynamic analysis. Hence, the six datasets we use reflect three types of features extracted using static analysis and three types of features extracted using dynamic analysis. Our goal is to set a new baseline in the field of Android malware classification and give guidance to practitioners on which techniques to use with certain types of datasets. To this end, we carry out a thorough evaluation of minority oversampling techniques.

Contributions. This paper makes the following specific contributions:

- We evaluate the effectiveness of 70 SMOTE variants for Android malware detection on a common benchmark. We use 4572 benign samples and 2399 malware

[§]Both authors have equal contributions in this work.

samples. Benign samples were randomly collected from Androzoo repository [19], which are released from year 2017 to 2019. 1208 malware samples are collected from Androzoo repository [19], which are from year 2017 and 2019, and 1191 malware samples are from Drebin repository [8]. We extract static features from call graph of Android package (apk) codes and dynamic features by executing the app in an Android emulator using our in-house intent-fuzzer combined with Android's Monkey testing framework [20].

- We extract 6 datasets that represent three types of features (*sequence*, *use*, and *frequency* of API calls) and two types of feature extraction techniques (*static* and *dynamic* analyses) commonly used in Android malware detection. We rank the SMOTE variants based on three evaluation criteria (precision, recall, F-measure) and determine the best suited variants on different types of features and feature extraction techniques used. And we provide meaningful insights into their performances.
- For open science and the benefit of the community, the script and the datasets used in this study are published in https://github.com/Jesper20/smote.

II. Related Work

A. Android Malware Classification

Our previous work [21] conducted an empirical evaluation to compare different types of features and classifiers for Android malware classification. We compared the performance of using *sequence* and *use* of API call features, which are extracted using static analysis and dynamic analysis. We also evaluated and compared the performances of conventional machine learning (ML) classifiers and deep learning (DL) classifiers. By contrast, this work focuses on the evaluation of various minority oversampling variants to enhance the performance of existing machine learning approaches and available datasets. On top of this, we present classification results with new data features, i.e., *frequency* of API calls, and a combination of all feature types, i.e., *sequence*, *use*, and *frequency* of API calls.

Features. There are several static analysis-based approaches that learn on sequence of API calls [3], [4], [22]–[24], use of API calls [5]–[8], [12], [15], or frequency of API calls [9], [10], [23], [25]. To extract static features from the Android app, call graphs [4], [12], data dependency graphs [15], [24], and control flow graph [26] have been used. On the other hand, dynamic analysis-based approaches, such as [2], [13], [27], learn on API call features extracted from execution traces. We should note that method-level analysis may provide datasets with millions of features, which are impractical in terms of training time and computing resource. Abstracting such features at levels like class, package, etc. can be done to speed up the machine learning training process [4], [11], [12].

The aforementioned approaches show that different types of features and feature extraction techniques may be used in Android malware classification. Since our evaluation consists of these commonly-used features, our work complements existing malware classification approaches in terms of dealing with imbalanced data.

B. Dealing with Imbalanced Data

In imbalanced learning, the focus is on improving general purpose classifiers when the data exhibits significant variations in terms of sample distributions between classes. Usually, this happens in areas where it is expensive or impossible to assemble large and balanced datasets, e.g., medical predictions, anomaly/malware detection, etc. In these domains, the number of positive samples, e.g., patients with cancer, is usually much smaller compared to that of negative samples, e.g., patients with no cancer. With imbalanced datasets, classifiers tend to overfit majority classes. To deal with this issue, undersampling or oversampling can be applied.

In undersampling, samples are removed from the majority class to address the imbalance. Undersampling can remove noises and provide a compact dataset which enables faster training. However, it may discard useful samples and increase the variance of the classifier [28]. When the imbalance ratio is large, undersampling will need to remove more training examples and, thus there may not be enough training data for the classifier to generalize properly. In this work, we focus on oversampling methods, which in general can address the class imbalance issue by generating more minority class examples, and at the same time retaining important training data for the majority class. The pioneering work by Chawla et al. [16] introduced the Synthetic Minority Oversampling Technique (SMOTE), which generates minority samples by randomly selecting the points along the line segments between neighboring minority instances.

Since its introduction, oversampling techniques have been highly successful in various ML applications. This success can be explained via two key factors: 1) oversampling is just a preprocessing step that can be applied before any ML techniques; and 2) oversampling addresses the root of data imbalanced problems, which is the lack of data. In this sense, oversampling is basically a form of data augmentation. Although there have been more than 100 SMOTE variants reported in the literature [29], not all of them are suitable for an automated empirical evaluation. In [17], the authors implemented and evaluated 85 different SMOTE variants with 104 imbalanced datasets. Most of these datasets have been collected from the Knowledge Extraction based on Evolutionary Learning (KEEL) repository (https://sci2s.ugr.es/keel/datasets.php). In contrast, our work focuses on evaluating and comparing different SMOTE variants in the unexplored domain of Android malware classification. It has been demonstrated that the performance

of SMOTE variants can vary significantly across datasets and application domains [17], [29].

There are a few approaches that make use of SMOTE techniques to tackle the imbalanced dataset problem in Android malware classification. For instance, in [30], Raff et. al considered one of the more popular SMOTE variants, namely Borderline-SMOTE, which improves SMOTE by limiting the oversampled points to data points close the border of the classes. In [31], the authors studied a dataset consisting of 5774 Android apps, among which 1582 are malware. They made use of the original SMOTE variant to produce a class balanced dataset and to improve the performance of their classifiers. [32] described the application of SMOTE, SVM based classifiers, and evolutionary algorithms for hyperparameter tuning in the detection of Android ransomware. The authors utilized a dataset consisting of 10,153 Android apps, in which 500 of them are ransomware. In [33], Dehkordy et al. balanced their dataset with undersampling, SMOTE, and a combination of both methods. They then applied KNN, SVM, and Iterative Dichotomiser 3 to create the classification model, and demonstrated that KNN+SMOTE outperformed other algorithms.

To the best of our knowledge, none of the existing works has conducted a thorough comparative study of majority of the available SMOTE variants in the domain of Android malware classification. In this paper, we aim to set a new baseline and provide practical guidance on the selection and application of different SMOTE variants in the area of Android malware classification.

III. Datasets

Motivated by our observation that there are typically three types of features (sequence of API calls, use of API calls, frequency of API calls) and two types of techniques (static analysis and dynamic analysis) used in Android malware classification, we use six different datasets in this study. Table I shows the characteristics of the datasets. *dsf* refers to the dataset containing sequence of API calls features extracted using dynamic analysis; *ssf* refers to the dataset containing sequence of API calls features extracted using static analysis; *duf* refers to the dataset containing use of API calls features extracted using dynamic analysis; *suf* refers to the dataset containing use of API calls features extracted using static analysis; *dfqf* refers to the dataset containing frequency of API calls features extracted using dynamic analysis; and *sfqf* refers to the dataset containing frequency of API calls features extracted using static analysis.

The first four of our datasets in Table I are the same ones used in our previous work [21]. The other two datasets are the new ones extracted using the same methodology described in [21]. For self-containment, we briefly describe the methodology below.

Static analysis and dynamic analysis are performed on a set of 6,971 benign and malware Android application packages (apks). Benign apks are obtained from Androzoo repository [19], which are released from year 2017 and 2019. Regarding malware, we obtained 1,187 malware apks from Drebin repository [8] and 1,212 malware apks from Androzoo repository, which are released from year 2017 and 2019. The datasets are extracted from those apks as follows:

Table I: Characteristics of datasets

Dataset	Type	Analysis	#benign	#malware	#features
dsf	Sequence	Dynamic	4572	2399	20000
ssf	Sequence	Static	4572	2399	85000
duf	Use	Dynamic	4572	2399	19357
suf	Use	Static	4572	2399	134558
dfqf	Frequency	Dynamic	4572	2399	19357
sfqf	Frequency	Static	4572	2399	134558

Static analysis. FlowDroid [34] is used to extract call graphs from apks. Call graphs represent the calling relationships between APIs.

Dynamic analysis. An intent fuzzer developed in [35] is used to generate test inputs for exploring the paths in call graphs. For more comprehensive code coverage and triggering possible abnormal app behaviors, Google's Android Monkey tool [20] is also used to randomly generate user interactions and UI input events such as tap, input text, etc.

The three types of features are then extracted from call graphs and execution traces. This results in three datasets reflecting sequence, use, and frequency of API calls, respectively, based on static analysis and three datasets reflecting sequence, use, and frequency of API calls, respectively, based on dynamic analysis. Feature extraction process is similar for both call graphs and execution traces.

Extracting sequence of API calls. Given a call graph, we traverse the graph in a depth first search manner and extract API calls as we traverse (hence, sequence). If there is a loop, the method is traversed only once. Similarly, we extract methods from execution traces. However, since execution traces are already sequences, depth first search is not necessary. Sample sequence of API calls extracted from a malware app $com.GoldDream.TingTing06ii$ is shown in Figure 1. And a sample dataset containing the *sequence features* is shown in Figure 2.

Extracting use of API calls. We initially build a database that stores unique classes. Given call graphs or execution traces, class signatures are extracted. Each unique class in our database corresponds to a feature[1]. The value of a feature is 1 if the corresponding class is found in the given call graph or execution trace; otherwise, it is 0. Figure 3 shows a sample dataset containing the *use features*.

[1]Note: the API calls that we extract here are abstracted at class level. This is because extracting features at method level will result in millions of features that will cost significantly huge training time. Our previous work [21] and other studies [4], [11] have observed that abstracted API calls features characterize Android malware even better and achieved better results.

Extracting frequency of API calls. In this study, we extract two new datasets with regards of frequency of API calls. Extracting these two datasets is similar to extracting the use of API calls datasets, except that, for each unique class signature, we record the number of class signature found in the given call graph or execution trace. Figure 4 shows a sample dataset containing the *freq features*.

```
android.content.Intent
android.telephony.PhoneStateListener
java.lang.StringBuilder
android.net.NetworkInfo
android.view.ContextThemeWrapper
android.widget.TextView
java.net.URI
java.nio.channels.SocketChannel
```

Figure 1: Sequence of API calls from a sample malware

	seq0	seq1	...	seq$n$	label
benign1	1590	1591	...	13	0
benign2	4379	4377	...	0	0
mal1	3	480	...	0	1
mal2	11907	307	...	0	1

Figure 2: Sample dataset containing the *sequence* features. Sequence length n is fixed at 20,000 for dynamic features and 85,000 for static features, which are the median lengths observed in our datasets.

	PhoneState Listener	Network Info	Telephony Manager	label
benign1	1	0	0	0
benign2	0	0	1	0
mal1	1	1	0	1
mal2	0	1	1	1

Figure 3: Sample dataset containing the *use* features

	PhoneState Listener	Network Info	Telephony Manager	label
benign1	0	2	0	0
benign2	0	0	2	0
mal1	11	7	0	1
mal2	0	0	23	1

Figure 4: Sample dataset containing the *frequency* features

All our datasets are of the same size containing 2,399 malware samples and 4,572 benign samples. Their imbalanced ratio IR can be computed as $IR = 4,572/2,399 = 1.9$

IV. Data Preprocessing & Classifiers

Oversampling. Before learning the classifiers on a given dataset, we run SMOTE and its variants to balance the dataset. SMOTE has more than 100 variants in the literature [29] but not all techniques have been well developed and implemented. In this study, we use the Python implementation of 85 oversampling methods by Kovács et al. [17]. Initially, we ran all the 85 oversampling methods implemented in [17] on two of our smallest datasets (*dsf* and *duf*). However, we encountered issues with 15 methods. Most of the issues are related to value errors (e.g. SSO, E_SMOTE, ISOMAP_Hybrid). It is likely that when new samples are generated, certain feature values may go outside the maximum possible integer value or may become null. Some methods take a very long time to complete (e.g. SMOTE_PSOBAT took 60 CPU hours for *dsf*). Hence, we had to omit those oversampling methods and used the remaining 70 methods for further experiments. Due to space constraints, we do not list those 70 oversampling methods here; instead they are listed in our GitHub page [36].

Data scaling. After oversampling, a standard data preprocessing method called standard scaler [37] is used to scale and transform each feature independently, so that its data distribution has a mean of zero and standard deviation of one. Data scaling mitigates the problem of different scales across features and makes the learning more efficient (the algorithm converges faster).

Classifiers are trained and tested on each dataset. In this study, we focus on conventional ML classifiers instead of deep learning classifiers; as the main goal is to see how different SMOTE variants can affect the classification performance. We plan to investigate the effect of SMOTE variants on deep learning classifiers in our future work. As there are many SMOTE variants, and some of them are very computationally expensive, e.g., they can take days to execute, we select two representative classifiers studied in [21], namely Naive Bayes and Random Forest. The first one has been shown to be very fast; while the second one was the top overall performer in the above mentioned study. We used Scikit-Learn Python library [38] to run the above classifiers with the library's default settings.

Random Forest, RF is an ensemble of classifiers using many decision tree models [39]. A different subset of training data is selected with a replacement to train each tree. The remaining training data serves to estimate the error and variable importance. RF has been proved to be highly accurate classifier for malware detection [40]. In our experiments, we used 10 classifiers to form an ensemble.

Naive Bayes, NB classifier applies Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable [41]. This assumption allows NB to learn the model extremely fast.

V. Evaluation

This section presents the experimental comparison results of 70 different SMOTE variants for Android malware classification. Specifically, we investigate the following research questions:

- **RQ1:** Which SMOTE variant is best suited for training with sequence of API calls features?
- **RQ2:** Which SMOTE variant is best suited for training with use of API calls features?
- **RQ3:** Which SMOTE variant is best suited for training with frequency of API calls features?
- **RQ4:** Which SMOTE variant is best suited for training with all three types of features?
- **RQ5:** How effective are the best SMOTE variants as compared against the classifiers trained without SMOTE?

A. Experiment Design

Datasets. We use the six datasets, discussed in Section III. The datasets include 4,572 benign apps and 2,399 malware. Our dataset is much larger than the ones used in recent studies [17], [18] that evaluate SMOTE variants in the context of defect prediction and comparable to the ones used in some recent Android malware classification studies such as [11], [24].

Performance measures. To evaluate the performance of SMOTE variants, we use Recall (probability of detection, Pd), Precision (Pr), and F-measure (F); which are standard measures typically used for evaluating malware detection accuracy [4], [10]. They are defined as follows:

- *Recall* $Pd = tp/(tp + fn)$
- *Precision* $Pr = tp/(tp + fp)$
- *F-measure* $F = 2 \times (Pr \times Pd)/(Pr + Pd)$.

where tp is the true positives (detected malware), fn is the false negatives (missed malware), and *fp* is the false positives (benign reported as malware). Recall measures the ability of the model to find all the relevant cases (true positives – malware) within a dataset. Precision measures the ability of the model to identify *only* the relevant cases. *F-measure* is the harmonic mean of precision and recall. It reports a balance between precision and recall.

In addition, we also collect the training time (T), time taken for oversampling, data scaling, and fitting the training samples, for each SMOTE variant.

Cross validation. We use stratified five-fold cross validation, a standard statistical analysis method [42], to evaluate the performances. Our cross validation process is as follows:

1) the given dataset (e.g. *dsf*) is randomly divided into five buckets and the buckets are randomly ordered.
2) each bucket is treated as a test set. For each test set, classifier is trained on the other four buckets (training set) and tested on the test set as follows:
   a) the training set is sampled using one of the oversampling methods
   b) standard scaler is applied to the training set
   c) a classifier (Naive Bayes or Random Forest) is learnt on the resulting training set
   d) standard scaler is applied to the test set
   e) the learnt classifier is applied to the scaled test set and the performance measures are computed.
3) the above two steps are repeated five times. The mean and standard deviation of the performance measures of all five trials are computed to make an evaluation.

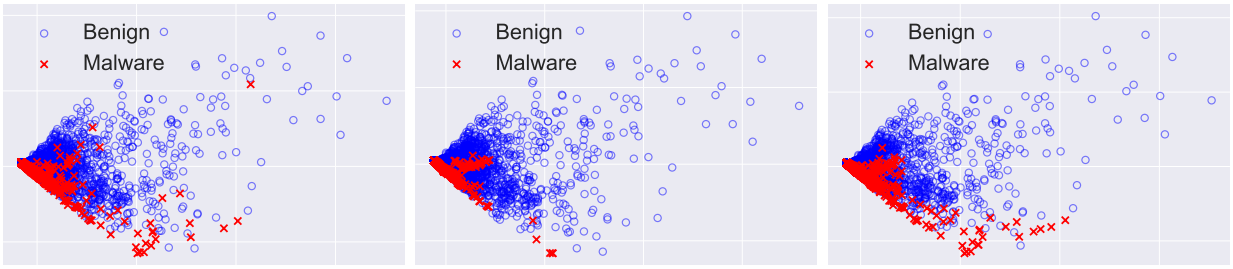
This process is done for all the 70 oversampling methods and for each classifier (Naive Bayes and Random Forest). For RQ5, we also run the experiments without an oversampling method (i.e., omitting the step 2a above). The following section reports the results for these variants.

For implementations of the above experiment design, we use Scikit-Learn [38] and Python libraries from [17]. The experiments were performed on a Linux machine with 40 cores Intel CPU E5-2640 2.40GHz and 330GB RAM. It took us about one week to extract the two new datasets from the 6,971 apks. It took us about 2 months to complete the experiments.

B. Results

RQ1: Best variants for sequence of API calls. We compare the performance of various SMOTE variants for dynamic and static datasets characterizing the sequence of API calls (the *dsf* and *ssf* datasets, respectively). In each table below, due to space constraints we only show the performance of selected variants, namely the top 5 and bottom 5 of all variants tested for a particular dataset. Tables II and III show the top and bottom 5 SMOTE variants for the classifiers using dynamic API sequence features. We observe that there are a great discrepancy between the top and the worst performing variants. In particular, for the NB classifier, NRAS significantly outperforms V_SYNTH in terms of F-measure. In addition, the standard deviations of the F-measures for the top performing variants are quite low, e.g., around 0.01. As expected, the RF classifier generally performs better than NB, except when unsuitable SMOTE variants are applied - for instance, the ADG variant results in the worst F-measure even when compared to NB with V_SYNTH. This highlights the importance of careful evaluations and selections of appropriate oversampling techniques for a specific problem domain.

Surprisingly, we also note that some selected variants which perform very well for one classifier can be negatively affecting another. For example, NRAS and ADG are the top-2 performers for NB, but the opposite happens when we apply them before training RF, as shown in Table III. To understand this better, we project the original data (dynamic-sequence features) and the oversampled data with NRAS onto a 2D plane using Principle Component Analysis [43]



(a) Original data (b) NRAS (c) SMOTE_ENN

Figure 5: Visualization of the original and datasets sampled with NRAS and SMOTE_ENN via Principle Component Analysis. The dataset shown here contains dynamic features characterizing the sequence of API calls (*dsf* dataset).

Table II: NB classifier with dynamic-sequence features

Variant	Pd	Pr	F	F (sd)
NRAS	0.973	0.398	0.565	0.015
ADG	0.973	0.395	0.561	0.015
Gazzah	0.982	0.391	0.559	0.013
SVM_balance	0.985	0.387	0.556	0.011
SMOTE_ENN	0.986	0.384	0.553	0.011
SOMO	0.988	0.382	0.551	0.01
SL_graph_SMOTE	0.993	0.368	0.537	0.007
VIS_RST	0.994	0.366	0.535	0.005
MDO	0.997	0.365	0.535	0.006
ISMOTE	0.892	0.386	0.526	0.032
V_SYNTH	0.459	0.268	0.282	0.224

Table III: RF classifier with dynamic-sequence features

Variant	Pd	Pr	F	F (sd)
SMOTE_ENN	0.778	0.57	0.657	0.036
Stefanowski	0.682	0.626	0.651	0.033
VIS_RST	0.926	0.496	0.645	0.022
SPY	0.677	0.613	0.641	0.067
Supervised_SMOTE	0.622	0.645	0.632	0.034
CURE_SMOTE	0.507	0.674	0.576	0.059
CCR	0.499	0.677	0.572	0.063
IPADE_ID	0.983	0.4	0.568	0.013
NRAS	0.462	0.683	0.546	0.073
ADG	0.102	0.313	0.132	0.109

Table IV: NB classifier with static-sequence features

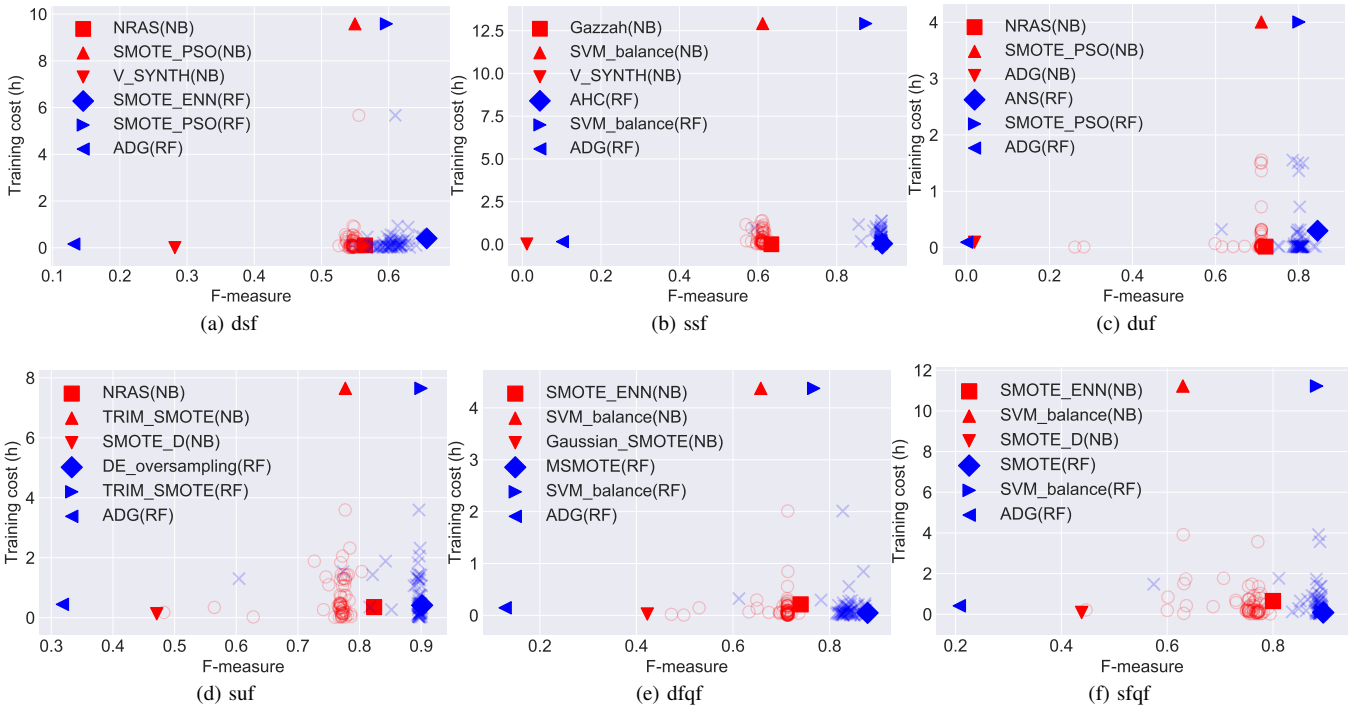
Variant	Pd	Pr	F	F (sd)
Gazzah	0.942	0.477	0.633	0.013
NRAS	0.972	0.463	0.627	0.015
CCR	0.977	0.455	0.620	0.018
CURE_SMOTE	0.979	0.453	0.619	0.013
SOMO	0.977	0.453	0.619	0.013
IPADE_ID	0.989	0.414	0.583	0.014
ADG	0.814	0.454	0.583	0.017
MDO	0.994	0.400	0.570	0.005
VIS_RST	0.994	0.398	0.569	0.008
V_SYNTH	0.006	0.090	0.011	0.008

Table V: RF classifier with static-sequence features

Variant	Pd	Pr	F	F (sd)
AHC	0.937	0.896	0.916	0.008
cluster_SMOTE	0.935	0.898	0.916	0.008
distance_SMOTE	0.930	0.901	0.916	0.006
ASMOBD	0.937	0.894	0.915	0.009
SN_SMOTE	0.933	0.896	0.914	0.007
SVM_balance	0.953	0.807	0.874	0.008
ISMOTE	0.944	0.793	0.862	0.018
VIS_RST	0.967	0.768	0.856	0.017
IPADE_ID	0.991	0.421	0.590	0.018
ADG	0.069	0.238	0.102	0.116

in Figure 5. NRAS, which is a sampling technique focusing on noise reduction and selective sampling of the minority class, does actually remove some data points from the malware class, as shown in Figure 5(b), before adding new synthetic ones. This mechanism appears to help NB, which is a simple classifier, performing better, but not for RF. On the other hand, SMOTE_ENN performs well for both NB and RF classifiers. This SMOTE variant uses Wilson's Edited Nearest Neighbor Rule (ENN) [44] to remove any training example whose class label differs from the class of at least two of its three nearest neighbors. The removed examples can be from both benign and malware classes. It is observed from Figure 5(c) that SMOTE_ENN does remove some noise while adding more data for the minority class, as compared to the original data.

As oversampling data before training classifiers may incur significant cost in terms of time, Fig 6a shows the F-measures and training costs for all the SMOTE variants tested with both the NB and RF classifiers, using the dynamic-sequence dataset (*dsf*). The notable cases are highlighted in the legend of this figure. In particular, we observe that using the best variant for each classifier, i.e., NRAS or SMOTE_ENN, incurs low training costs. This reaffirms the practicality of using appropriate oversampling methods in classifier training with imbalanced datasets. At the same time, we note that some particular SMOTE variants could be too computationally expensive to be used in Android malware classification, e.g., SMOTE_PSO which employs Particle Swarm Optimization (PSO) to do oversampling.



(a) dsf (b) ssf (c) duf

(d) suf (e) dfqf (f) sfqf

Figure 6: F-measures versus training cost for all SMOTE variants applied to each of the datasets used. We show the legends for selected variants only (best/worst F-measures, and highest cost). Blue/red data points are for RF and NB, respectively.

Tables IV and V show the obtained results on recall, precision, F-measure for the top-5 and bottom-5 SMOTE variants when using the static features characterizing sequence of API calls (the *ssf* dataset). Similarly, Figure 6b shows the training cost versus the F-measure for all SMOTE variants tested for this dataset. We observe some familiar names here, such as NRAS as a good performer, while V_SYNTH and ADG as performing poorly for this dataset. The combination of either AHC (agglomerative hierarchical clustering) oversampling, cluster_SMOTE, or distance_SMOTE; and RF classifier provides very good F-measure (0.916) with low standard deviations and training time without any fine tuning.

When we compare the results shown in Figures 6a and 6b, it is noted that the static-sequence features provide a markedly improved F-measure compared to that of the dynamic-sequence features. This improvement applies to both NB and RF classifiers, and at the same time, it significantly widens the performance gap between NB and RF. We might partly attribute such improvement to the fact that the *ssf* dataset has about 4 times more features compared to the *dsf* dataset as shown in Table I; this also explains the increase in training cost for the former. We also observe that when using API sequence based features, the performance and training cost for the majority of SMOTE variants are quite similar; this is shown by the two distinct clusters for NB and RF, respectively in Figures 6a and 6b.

Summary-RQ1: For sequence of API calls, NRAS and Gazzah are the best for NB; while SMOTE_ENN and AHC are the best for RF. Notably, SMOTE_ENN performs quite well for both NB and RF. It is not possible to identify a single variant that performs well in both static and dynamic datasets for RF, while we can observe that variants based on noise reduction techniques, e.g., NRAS, SMOTE_ENN, etc., are generally good for NB.

RQ2: Best variants for use of API calls. Figures 6c and 6d, as well as Tables VI-IX summarize experiment results when using all SMOTE variants to train our two classifiers with the use of API features (*duf* and *suf* datasets). Similar to the previous section, we observe that popular variants such as NRAS and SMOTE_ENN are among the top performers in terms of F-measure for the NB classifier, while ADG and V_SYNTH do not lead to good classification performance. For the RF classifier, it appears that there is no SMOTE variant that consistently performs well - we see a different top variant for each dataset used. However, we can identify ADG as the variant that consistently results in poor classification performance for the RF classifier across different datasets.

When compared to the previous experiment which uses API sequence features, we observe that the SMOTE

variants tested with API use features produce a more varying level of classification performance, as shown in Figures 6c and 6d. This emphasizes the importance of choosing the appropriate oversampling method when one has to do malware classification with just the data on API use. In addition, for this type of data, RF is still the better classifier overall, but the performance gap between RF and NB is not as pronounced as seen in the previous experiment.

Summary-RQ2: For use of API calls, NRAS is the best for NB; while ANS, DE_oversampling, and the original SMOTE variant are the best for RF. Notably, noise reduction-based approaches like NRAS and SMOTE_ENN generally perform quite well for NB in both static and dynamic datasets. It is not possible to identify a single variant that performs well in both datasets for RF.

Table VI: NB classifier with dynamic-use features

Variant	Pd	Pr	F	F (sd)
NRAS	0.845	0.635	0.720	0.099
ISMOTE	0.879	0.612	0.718	0.089
SMOTE_ENN	0.855	0.619	0.713	0.103
Safe_Level_SMOTE	0.932	0.576	0.711	0.026
SMOTE_TomekLinks	0.857	0.614	0.711	0.105
MDO	0.849	0.484	0.615	0.115
CCR	0.778	0.509	0.599	0.243
Gaussian_SMOTE	0.220	0.407	0.283	0.150
SMOTE_D	0.202	0.380	0.261	0.156
ADG	0.011	0.072	0.019	0.039

Table VII: RF classifier with dynamic-use features

Variant	Pd	Pr	F	F (sd)
ANS	0.856	0.837	0.845	0.044
AND_SMOTE	0.877	0.815	0.841	0.032
Safe_Level_SMOTE	0.890	0.792	0.835	0.028
SMOTE_TomekLinks	0.832	0.809	0.813	0.061
SMOTE_RSB	0.846	0.794	0.813	0.070
SMOTE_D	0.760	0.828	0.785	0.088
VIS_RST	0.975	0.616	0.753	0.036
NEATER	0.689	0.837	0.750	0.102
IPADE_ID	0.997	0.445	0.614	0.035
ADG	0.000	0.040	0.001	0.002

RQ3: Best variants for frequency of API calls. Figures 6e and 6f summarize the experiment results when using all SMOTE variants to train our two classifiers with the frequency of API call features (the *dfqf* and *sfqf* datasets). We do not include the tables of top-5 and bottom-5 variants here due to space constraints. We observe similar results to the previous experiments. In particular, noise reduction-based variants such as NRAS and SMOTE_ENN generally perform well for the NB classifier, while the RF classifier works better with one particular variant for a different dataset. Notably, the original SMOTE [16] method produces the highest F-measure for RF when using static-frequency features. Similar to what have been observed in other datasets, the ADG variant performed poorly here as well.

Table VIII: NB classifier with static-use features

Variant	Pd	Pr	F	F (sd)
NRAS	0.956	0.724	0.824	0.011
SMOTE_ENN	0.945	0.699	0.804	0.013
OUPS	0.970	0.661	0.786	0.008
SOMO	0.970	0.660	0.786	0.010
MSYN	0.970	0.658	0.784	0.007
VIS_RST	0.985	0.576	0.727	0.010
V_SYNTH	0.985	0.461	0.628	0.015
CCR	0.514	0.690	0.564	0.142
Gaussian_SMOTE	0.365	0.712	0.483	0.039
SMOTE_D	0.356	0.696	0.471	0.029

Table IX: RF classifier with static-use features

Variant	Pd	Pr	F	F (sd)
DE_oversampling	0.955	0.854	0.902	0.013
SMOTE	0.951	0.857	0.901	0.013
NDO_sampling	0.948	0.859	0.901	0.013
SMOTE_Cosine	0.950	0.857	0.901	0.012
SMOTE_OUT	0.946	0.860	0.901	0.013
NEATER	0.756	0.904	0.821	0.033
NRAS	0.745	0.911	0.817	0.055
SMOTE_ENN	0.682	0.900	0.773	0.049
IPADE_ID	0.994	0.435	0.604	0.033
ADG	0.600	0.229	0.318	0.258

Regarding training cost, it is observed that computationally expensive variants such as SVM_balance do not necessarily produce the best classification performance. We also note that the training cost is much higher in the *sfqf* dataset as compared to the *dfqf* dataset. This is because the static dataset has many more features (about 135k as compared to 19k). Despite having more features, the static dataset here does not produce much better classification performance. Furthermore, all SMOTE variants tested result in more varying F-measure values when using frequency based datasets, as compared to the API sequence datasets. This is illustrated in Figures 6e and 6f.

Summary-RQ3: For frequency of API calls, SMOTE_ENN is the best for NB; while MSMOTE and the original SMOTE are the best for RF. It is not possible to identify a single variant that performs well in both static and dynamic datasets for RF, while we can observe that variants based on noise reduction techniques, e.g., NRAS, SMOTE_ENN, etc., are good for NB.

RQ4: Using a combined dataset. In this experiment, we combine all feature types, namely API use, API sequence and API frequency; as well as two types of program analyses, namely static and dynamic analysis, into one single dataset. Due to the sheer number of features and therefore training time, for this dataset, we choose to run only selected SMOTE variants that perform well in terms of F-measure in our previous experiments. They are: NRAS, SMOTE_ENN, Gazzah, AHC, ANS, DE_oversampling,

MSMOTE, and last but not least, the original SMOTE. We only summarize the notable results from this experiment here, due to space constraints. We note that combining all feature types and analyses does increase the training time substantially, but it does not improve the classification performance correspondingly. For instance, RF(AHC) provides the best F-measure of 0.911 (standard deviation of 0.022) when using this combine dataset, compared to a previously obtained best value of 0.916 (standard deviation of 0.008) when using the *ssf* dataset. Although in this case AHC and NRAS are still the best SMOTE variants for RF and NB respectively, we believe that it is more useful for practitioners of Android malware classification to identify the right types of features and program analyses before applying SMOTE variants to resample any imbalanced datasets.

Summary-RQ4: The classification performance does not improve when using all feature types and program analyses in a combined dataset for the best SMOTE variants.

RQ5: Comparing to non-SMOTE classifiers. Table X summarizes the results when comparing non-SMOTE classifiers against the best SMOTE variants selected from our previous experiments. We observe that for each classifier, the SMOTE variant helps improve the F-measure. The improvement is statistically significant as confirmed by the Wilcoxon signed rank test (the two-tailed p-value is calculated as 0.00222).

Table X: Comparing classifiers with and without the best SMOTE variants. The best F-measures for NB and RF are highlighted with the corresponding training costs.

Classifier	Data	Pd	Pr	F	F(std)	Cost(h)
NB (NRAS)	dsf	0.973	0.398	0.565	0.015	0.102
RF (SMOTE _ENN)	dsf	0.778	0.57	0.657	0.036	0.4
NB	dsf	0.991	0.376	0.546	0.016	0.03
RF	dsf	0.55	0.691	0.605	0.171	0.257
NB (Gazzah)	ssf	0.942	0.477	0.633	0.013	0.006
RF (AHC)	ssf	0.937	0.896	0.916	0.008	0.042
NB	ssf	0.989	0.441	0.61	0.026	0.01
RF	ssf	0.92	0.905	0.913	0.01	0.16
NB (NRAS)	duf	0.845	0.635	0.72	0.099	0.018
RF (ANS)	duf	0.856	0.837	0.845	0.044	0.3
NB	duf	0.857	0.613	0.71	0.209	0.002
RF	duf	0.807	0.83	0.817	0.084	0.004
NB (NRAS)	suf	0.956	0.724	0.824	0.011	0.354
RF (DE _oversampling)	suf	0.955	0.854	0.902	0.013	0.414
NB	suf	0.968	0.675	0.795	0.005	0.012
RF	suf	0.924	0.879	0.901	0.028	0.012
NB (SMOTE _ENN)	dfqf	0.867	0.652	0.74	0.068	0.212
RF (MSMOTE)	dfqf	0.859	0.901	0.879	0.032	0.048
NB	dfqf	0.847	0.621	0.712	0.102	0.008
RF	dfqf	0.776	0.899	0.827	0.084	0.004
NB (SMOTE _ENN)	sfqf	0.937	0.703	0.8	0.041	0.648
RF (SMOTE)	sfqf	0.934	0.864	0.895	0.034	0.084
NB	sfqf	0.957	0.665	0.783	0.072	0.018
RF	sfqf	0.923	0.861	0.888	0.069	0.014

Table X also shows the comparison of training times and F-measures obtained for NB and RF, with and without applying the selected SMOTE variants. Most of the SMOTE variants increase the cost in terms of training time for the classifiers. This is expected as resampling the original datasets requires additional computation to generate more samples. In some cases, e.g., NRAS, SMOTE_ENN, etc., the computation time could be much more due to the required calculations for removing noisy samples in the original dataset, before adding new synthetic samples. Notably, some variants such as AHC, Gazzah, etc. actually reduce the training time significantly due to their very fast sampling time, and at the same time, improve the classification performance with the *ssf* dataset, as indicated in Table X. It is interesting to see that the best performing classifier, RF with AHC, does have the shortest training time together with the best F-measure overall.

Summary-RQ5: The best SMOTE variants tested in this study statistically significantly improve the performance of Android malware classifiers.

C. Threats to Validity

Like any empirical study, biases may affect the generalizability of our results.

Evaluation bias. In terms of training and testing, certain ordering and splitting of samples may affect the results. To reduce this bias, we use five-fold cross validation, which randomizes the order of the groups and the sampling process. Five-fold cross validation was also used in several malware and defect prediction studies [3], [18], [45]. Furthermore, we also used stratified sampling which keeps the imbalanced ratio of the total dataset size.

In terms of performance metrics, defect prediction studies such as [17], [18] used Area Under Curve (AUC) to evaluate the SMOTE variants. But we used F-measure to rank the SMOTE variants because it is widely used in Android malware classification approaches, e.g., [3], [4], [10], [12]. F-measure reports a balance between recall and precision whereas AUC focuses on true positive rate versus false positive rate. In our experiments, we computed AUC scores as well but did not report here due to space constraints. Interested readers may refer to our website [36].

Sampling bias. Our datasets may not be representative of Android malware and benign samples in the wild. To reduce this bias, a) we analyzed real world Android apks (both benign and malware); b) we randomly collected the samples released in years 2017 to 2019; and c) we used a large sample size (larger than the ones used in related work [17], [18], [46], [47]).

The imbalanced ratio of our datasets (1.9) may not reflect the ratio of datasets used in training actual malware predictors. This is challenging because different malware classification approaches have used different imbalanced

ratios. For example, [3], [8], [11], [12], [15] used different imbalanced ratios of 22, 6, 0.35, 1.15, 2.1, respectively, whereas some other studies have used balanced datasets [24], [48]. As future work, we aim to evaluate the performance of SMOTE variants when trained with varying dataset sizes.

VI. Conclusion

This work addresses the problem of finding best suited oversampling methods for Android malware detection. Oversampling techniques are used for addressing data imbalanced problem and it is known that their performances vary across different application domains. In the evaluation, we used six different datasets that reflect two kinds of analyses and three types of features, which are commonly used in Android malware detection. The datasets were extracted from a common benchmark of 4,572 benign apps and 2,399 malware apps. Among other findings, we observed that oversampling could improve the F-measure of malware classifier by 3.16% on average. The improvement is most significant when training with sequence-type features. We also identified 8 oversampling methods that produce consistently good results across different datasets. Lastly we observed that sophisticated oversampling methods such as SVM_balance come at the cost of significantly longer training time (in the magnitude of hours); yet they do not necessarily perform better than simpler methods. In future work, we will investigate the performance of SMOTE variants when applied to other types of classifiers such as those based on deep learning techniques. We also plan to use a larger dataset and experiment with varying imbalanced ratios.

References

[1] Symantec, "Internet Security Threat Report," https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf, 2019.

[2] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in *Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2016, pp. 577–582.

[3] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.

[4] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Transactions on Privacy and Security*, vol. 22, no. 2, p. 14, 2019.

[5] A. Sharma and S. K. Dash, "Mining api calls and permissions for android malware detection," in *International Conference on Cryptology and Network Security*. Springer, 2014.

[6] P. P. Chan and W.-K. Song, "Static detection of android malware by using permissions and api calls," in *International Conference on Machine Learning and Cybernetics*, vol. 1. IEEE, 2014, pp. 82–87.

[7] S. Y. Yerima, S. Sezer, and I. Muttik, "High accuracy android malware detection using ensemble learning," *IET Information Security*, vol. 9, no. 6, pp. 313–320, 2015.

[8] D. Arp, M. Spreitzenbarth, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." 2014.

[9] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.

[10] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 3, p. 11, 2018.

[11] W. Yang, M. Prasad, and T. Xie, "Enmobile: Entity-based characterization and analysis of mobile malware," in *International Conference on Software Engineering*. IEEE, 2018, pp. 384–394.

[12] M. Ikram, P. Beaume, and M. A. Kaafar, "Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling," *arXiv preprint arXiv:1905.09136*, 2019.

[13] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: a multi-level anomaly detector for android malware," in *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*. Springer, 2012, pp. 240–253.

[14] W. Wang, Y. Li, X. Wang, J. Liu, and X. Zhang, "Detecting android malicious apps and categorizing benign apps with ensemble of classifiers," *Future Generation Computer Systems*, vol. 78, pp. 987–994, 2018.

[15] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *ACM SIGSAC conference on computer and communications security*, 2014, pp. 1105–1116.

[16] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[17] G. Kovács, "An empirical comparison and evaluation of minority oversampling techniques on a large number of imbalanced datasets," *Applied Soft Computing*, vol. 83, pp. 105–662, 2019.

[18] A. Agrawal and T. Menzies, "Is" better data" better than" better data miners"?" in *International Conference on Software Engineering*. IEEE, 2018, pp. 1050–1061.

[19] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *International Conference on Mining Software Repositories*. ACM, 2016, pp. 468–471.

[20] Android, "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey, 2019.

[21] L. K. Shar, B. F. Demissie, M. Ceccato, and W. Minn, "Experimental comparison of features and classifiers for android malware detection," in *International Conference on Mobile Software Engineering and Systems*. ACM, 2020, pp. 50–60.

[22] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé *et al.*, "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 301–308.

[23] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaminglized machine learning-based system for detecting android malware," in *Asia Conference on Computer and Communications Security*, 2016, pp. 377–388.

[24] F. Shen, J. Del Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek, "Android malware detection using complex-flows," *IEEE Transactions on Mobile Computing*, vol. 18, no. 6, pp. 1231–1245, 2018.

[25] M. Fan, J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, and T. Liu, "Frequent subgraph based familial classification of android malware," in *International Symposium on Software Reliability Engineering*. IEEE, 2016, pp. 24–35.

[26] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *IEEE Symposium on Security and Privacy*. IEEE, 2005, pp. 32–46.

[27] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Computing*, vol. 20, no. 1, pp. 343–357, 2016.

[28] A. Dal Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi, "Calibrating probability with undersampling for unbalanced classification," in *2015 IEEE Symposium Series on Computational Intelligence*. IEEE, 2015, pp. 159–166.

[29] A. Fernández, S. Garcia, F. Herrera, and N. V. Chawla, "Smote for learning from imbalanced data: progress and challenges, marking the 15-year anniversary," *Journal of artificial intelligence research*, vol. 61, pp. 863–905, 2018.

[30] E. Raff and C. Nicholas, "Malware classification and class imbalance via stochastic hashed lzjd," in *ACM Workshop on Artificial Intelligence and Security*, 2017, pp. 111–120.

[31] A. Tirkey, R. K. Mohapatra, and L. Kumar, "Anatomizing android malwares," in *Asia-Pacific Software Engineering Conference*. IEEE, 2019, pp. 450–457.

[32] I. Almomani, R. Qaddoura, M. Habib, S. Alsoghyer, A. Al Khayer, I. Aljarah, and H. Faris, "Android ransomware detection based on a hybrid evolutionary approach in the context of highly imbalanced data," *IEEE Access*, 2021.

[33] D. T. Dehkordy and A. Rasoolzadegan, "A new machine learning-based method for android malware detection on imbalanced dataset," *Multimedia Tools and Applications*, pp. 1–22, 2021.

[34] S. Arzt et al., "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, pp. 259–269.

[35] B. F. Demissie, M. Ceccato, and L. K. Shar, "Security analysis of permission re-delegation vulnerabilities in android apps," *Empirical Software Engineering*, pp. 5084–5136, 2020.

[36] L. K. Shar, "Evaluating oversampling techniques," https://github.com/Jesper20/smote, 2021.

[37] J. Han, M. Kamber, and J. Pei, "Data mining concepts and techniques third edition," *The Morgan Kaufmann Series in Data Management Systems*, vol. 5, no. 4, pp. 83–124, 2011.

[38] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[39] I. Barandiaran, "The random subspace method for constructing decision forests," *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 20, no. 8, pp. 1–22, 1998.

[40] M. Eskandari and S. Hashemi, "A graph mining approach for detecting unknown malwares," *Journal of Visual Languages & Computing*, vol. 23, no. 3, pp. 154–162, 2012.

[41] H. Zhang, "The optimality of naive bayes," *AA*, vol. 1, no. 2, p. 3, 2004.

[42] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[43] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[44] D. L. Wilson, "Asymptotic properties of nearest neighbor rules using edited data," *IEEE Transactions on Systems, Man, and Cybernetics*, no. 3, pp. 408–421, 1972.

[45] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in *Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2016, pp. 577–582.

[46] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," in *International joint conference on neural networks*. IEEE, 2008, pp. 1322–1328.

[47] W. Siriseriwan and K. Sinapiromsaran, "Adaptive neighbor synthetic minority oversampling technique under 1nn outcast handling," *Songklanakarin J. Sci. Technol*, vol. 39, no. 5, pp. 565–576, 2017.

[48] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 371–372.