# On the Impact of Sample Duplication in Machine Learning based Android Malware Detection

YANJIE ZHAO, Monash University, Australia
LI LI*, Monash University, Australia
HAOYU WANG, Beijing University of Posts and Telecommunications, China
HAIPENG CAI, Washington State University, United States
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg
JACQUES KLEIN, University of Luxembourg, Luxembourg
JOHN GRUNDY, Monash University, Australia

Malware detection at scale in the Android realm is often carried out using Machine learning techniques. State-of-the-art approaches such as DREBIN and MaMaDroid are reported to yield high detection rates when assessed against well-known datasets. Unfortunately, such datasets may include a large portion of duplicated samples, which may bias recorded experimental results and insights. In this paper, we perform extensive experiments to measure the performance gap that occurs when datasets are de-duplicated. Our experimental results reveal that duplication in published datasets has a limited impact on supervised malware classification models. This observation contrasts with the finding of Allamanis on the general case of machine learning bias for big code. Our experiments, however, show that sample duplication more substantially affects unsupervised learning models (e.g. malware family clustering). Nevertheless, we argue that our fellow researchers and practitioners should always take sample duplication into consideration when performing machine learning (via either supervised or unsupervised learning) based Android malware detections, no matter how significant the impact might be.

**ACM Reference Format:**
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 Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2021), 36 pages. https://doi.org/10.1145/3446905

## 1 INTRODUCTION

Security of mobile apps is now a critical research and practice issue. Mobile apps are used pervasively, including for critical activities such as transportation (e.g., smart car apps), finance (e.g., banking apps), and healthcare (e.g., heart rate monitoring apps). Even leisure apps, which are seldom viewed as critical, may pose security threats given that they can provide attackers easy access to sensitive information on users' devices [9, 39]. The diversification of app developers, vendors, and brokers

*Li Li is the corresponding author.

Authors' addresses: Yanjie Zhao, yanjie.zhao@monash.edu, Monash University, Australia; Li Li, li.li@monash.edu, Monash University, Australia; Haoyu Wang, Beijing University of Posts and Telecommunications, China, haoyuwang@bupt.edu.cn; Haipeng Cai, Washington State University, United States, haipeng.cai@wsu.edu; Tegawendé F. Bissyandé, University of Luxembourg, Luxembourg, tegawende.bissyande@uni.lu; Jacques Klein, University of Luxembourg, Luxembourg, jacques.klein@uni.lu; John Grundy, Monash University, Australia, john.grundy@monash.edu.


1049-331X/2021/1-ART1 $15.00
https://doi.org/10.1145/3446905

creates a great challenge to ensure that each app can be rapidly and effectively vetted from the perspective of security and privacy concerns.

Ideally, mobile apps should be intensively analyzed to check their security and privacy requirements conformance. However, when performed statically, app analysis is often time-consuming, may produce many false positives, and will not identify all problems that occur at runtime [26, 42]. On the other hand, dynamic analysis does not scale and often cannot cover the whole codebase [19, 34, 45]. In recent years the research community has progressively shifted to view machine learning (ML) as an affordable and worthwhile effort for identifying security issues, in particular, malicious behavior, in mobile apps [41].

An increasingly large body of research on machine learning-based approaches for predicting Android malware has been published. DREBIN [8] and MamaDroid [46] are state-of-the-art sample approaches that are commonly referred to. However, after promising results have been reported, the community has started to reflect on the potential biases that many machine learning-based research experiments carry. For example, Allix et al. [5] and then Pendlebury et al. [52] have experimentally shown that the performance of malware detectors is actually highly dependent on experimental parameters, such as dataset construction (e.g., spatial and temporal dimensions) or evaluation methodology (e.g., 10-fold cross validation). Dataset is a critical ingredient in the training and validation of all machine learning-based models. Nevertheless, to date the app analysis community has paid little attention to the intrinsic quality of datasets beyond the problems of class imbalance and temporal alignment [5, 52].

Concerning the quality of datasets and their impact on machine learning models, Allamanis [4] has recently raised the concern of *code sample duplication*, i.e., the phenomenon where a given sample is repeated several times in the dataset. This study reported that performance metrics of machine learning models for big code are sometimes inflated by up to 100% when testing on duplicated code corpora, compared to their performance on de-duplicated corpora. We consider this alarming finding to be relevant for further investigation in the field of ML-based Android malware detection since Android samples may also contain duplicated features, which are often extracted from Android apps' code snippets and metadata such as permissions or resource files. Indeed, it casts doubts on threats to validity on all research achievements in this area. Our work, in this paper, echoes this concern and attempts to clarify the effect of sample duplication on the performance of malware detectors.

Our investigations start with a review of recent state-of-the-art approaches in ML-based malware detection. Through this we identify some artifacts whose duplications in the datasets may not be obvious to experimenters. We then discuss theoretically the possible effect of sample duplication on learning models. Then, we quantify the extent of sample duplication in widely-used app malware datasets. Finally, we perform large scale experimental analyses of the effect of duplication by considering two learning scenarios for malware detection – binary classification of maliciousness as a supervised learning problem, and malware family clustering as an unsupervised learning problem. Based on our experiment findings, we provide a discussion on the quality of the current malware datasets as well as on the validity of their recorded performance in the literature.

Our experimental exploration reveals that (1) widely-used malware datasets include a considerable amount of duplicate data samples. Given recent studies on the adverse effect of duplicate code for machine learning models of code, many promising results and findings in the literature are potentially threatened by this issue of duplication bias. With comprehensive experiments covering different types of dataset samples, we have shown that (2) the impact of duplicates in commonly-used datasets remains marginal. This is for the typical supervised learning models that are proposed for app malware detection, no matter 10-fold cross-validation or in-the-wild experiments are applied. (3) the marginal impact is consistent across different machine learning models trained on different algorithms including RandomForest and SVM. (4) We have shown that unlike supervised learning for which insignificant impact is observed for malware detection, the impact of sample duplication on unsupervised learning (especially on malware clustering) is however quite significant. Based on these empirical findings, no matter how significant the impact of sample duplication might be, we appeal to the community that our fellow researchers and practitioners should always take sample duplication into consideration when performing future machine learning (via either supervised or unsupervised learning) based Android malware detections.

The remainder of this paper is organized as follows. Section 2 explains the problem scope of this work and Section 3 presents a preliminary study regarding the duplication phenomenon in publicly released Android malware datasets. After that, Section 4 presents our experimental design while Section 5 reports the corresponding experimental results. Later, Section 6 discusses the possible implications of this work and its potential limitations. Section 7 discusses the related work and finally Section 8 concludes this paper.

## 2 PROBLEM SCOPING

We provide key background information for helping readers better understand our study. Notably, we recall the main usage scenarios of machine learning in the field of malware detection and discuss different levels of app details that are recurrently leveraged to constitute learning datasets. Finally, we introduce the duplication bias problem.

### 2.1 Machine Learning based Malware Detection

Machine learning is relied upon to perform malware detection at a large scale in many published toolsets and experiments. There are mainly two scenarios: *binary classification* models are trained to predict the maliciousness of an app. These are generally a *supervised learning* scenario where the entire dataset is labeled for the experiments. In contrast, the problem of malware family identification is often modeled as an *unsupervised learning* scenario, i.e., samples are grouped together based on their similarity.

Table 1 enumerates a few examples of key published work in the field of machine learning based malicious behaviour analysis. A key observation from this table is that for the large majority of approaches, feature engineering targets **code** artifacts. Our study will thus focus on code-related samples.

For the purpose of our study, we focus on four levels of artifacts that may be subject to duplication when they constitute the learning datasets: (1) the **APK** samples (i.e., the whole app packages), (2) the **DEX** code (i.e., the entire code within the app package), (3) the **Opcode Sequence** (i.e., the low-level machine language instructions), and (4) the **API calls** (i.e., only the specific parts of the code that interact with the framework and access sensitive physical resources).

In practice, datasets used for learning are composed of samples of each of the aforementioned types and can include duplicates. This may subsequently lead to duplication bias (cf. Section 2.2).

Table 1. Sample approaches involving machine learning for malicious Android apps analyses.

| Year | Venue | Approach | features | Detection scenario |
|---|---|---|---|---|
| 2019 | TASE | CDGDroid [69] | CFG, DFG | Family classification |
| 2018 | TOSEM | RevealDroid [27] | API calls, reflection, native binaries | Bi- and family classification |
| 2017 | CODASPY | McLaughlin et al. [47] | Opcode sequence | Bi-classification |
| 2016 | NDSS | MaMaDroid [46] | API calls | Bi-classification |
| 2016 | IST | Wu et al. [63] | API calls | Bi-classification |
| 2015 | ICSE | MUDFLOW [11] | Sensitive data flows | One-class classification |
| 2015 | ICSE | Holland et al. [28] | API calls, permissions | Bi-classification |
| 2014 | NDSS | DREBIN [8] | API calls, permissions, etc. | Bi-classification |
| 2014 | SIGCOMM | Droid-Sec [68] | API calls, permissions, dynamic behavior | Bi-classification |
| 2013 | SecureComm | DroidAPIMiner [2] | API calls | Bi-classification |

**APK.** Android apps are distributed in the form of packages (APKs). Every app version has a distinct APK file. Malware datasets shared in the literature are generally formed by collecting APKs from a variety of sources such as the official Google Play store and other alternative third-party markets. The state-of-the-art machine learning based malware classifiers are usually proposed to flag Android malware at the APK level.

**DEX.** DEX file (i.e., Dalvik Executable) is the core file of an Android app that contains the actual programming code to be executed on a hosting Android device OS. By default, the DEX file is named as *classes.dex* in given Android apps. In this work, we consider DEX duplication exists in a dataset as long as the same DEX file appears in different apps (i.e., the hash value of the *classes.dex* file is identical between two different apps). For example, in the Drebin dataset, there are 128 different app samples that share the same package name, namely com.soft.android.appinstaller, for which their DEX hashes are the same despite their APK hashes being different.

**Opcode Sequence.** Bilar et al. [13] assert that opcodes (which can be disassembled from DEX files) can act as a predictor, since the distribution of malware opcode frequencies significantly differs from that of non-malicious software. In our manual observation, we found that different DEX files can indeed result in identical opcode sequences. These can be fulfilled by, for example, altering only the resource files of given Android apps, where some resources files in Android are used to set the constant values of certain variables or define the names of specific widgets. This type of change will lead to different DEX files but will not impact the actual code compiled to DEX files. Thereby, they will not impact the disassembled opcode sequences. In this work, we consider opcode sequence duplication to exist as long as different apps share the same opcode sequence. This is no matter whether the corresponding DEX files are identical or not. As an example, there are three app samples, namely com.brianrileyar.girlonfire, com.brianrileyar.fieldrunners, and com.brianrileyar.sonic, sharing the same opcode sequences but with different DEX hashes.

**API Call.** We go one step further to identify more fine-grained duplications in malware datasets. To this end, we look at the API calls (which can be extracted from an app's opcode) of Android apps, since APIs are one of the most important constituent parts of the apps. Even if two apps have different opcode sequences, these two apps may still have the same sequence of API calls, which may in turn lead to identical features when APIs are exclusively considered. In this work, we consider API call duplication to exist as long as there are a number of apps that access into the same number of APIs, and each API is called in the identical sequence and times. For example, in the Drebin dataset, the samples with package names, com.evilsunflower.reader.evilShenger, com.evilsunflower.reader.evilQichang, com.evilsunflower.reader.evilGuigu, and com.evilsunflower.reader.evilLiangxing, share the same API calls, although their opcode sequences are different.

### 2.2 Duplication in Machine Learning Datasets

Machine learning experiments require datasets to train models that provide recommendations on *new* and *unseen* samples. The standard expectation is that the models will *generalize* well to those new samples: the training process faithfully models the true distribution of the data as it will be observed by the particular application execution scenario. As discussed by Allamanis [4], in order for the machine learning model to generalize to the true data distribution[1], it needs to be trained on data independently drawn from that distribution. Unfortunately, sample duplicates commonly violate this assumption with varying consequences as duplicated samples will not introduce new knowledge to the learning models.

Sample duplication, in the domain of malware detection, refers to the idea that some learning data samples (e.g., the APK, the DEX code, etc.) appear multiple times within a corpus. The feature vectors subsequently extracted from the duplicated samples will likely be duplicated as well. Such duplication creates an issue as it biases the data distribution. Actually, a common practice in machine learning experiments is to split any existing dataset into two parts: a training set that is used to train the machine learning model and a test set where the performance of the model is measured. Since duplicated datasets are randomly distributed to the training set, the algorithms tend to learn different probability distributions, which may result in different results. Moreover, the splitting process may put the same samples (e.g., duplicated ones) into both training set and test set, leading also to biased learning.

**Definitions:** Assume a dataset $D$ of app information samples that is split into a training and a test set. Conceptually, we can distinguish three types of duplicates: (1) "in-train" duplicates, i.e., samples duplicated within the training set; (2) "in-test" duplicates, i.e., duplicates within the test set; and (3) "cross-set" duplicates, i.e., samples that appear both in training and test sets.

We borrow the terms of Allamanis to define the **Duplicate bias** [4]:

> In machine learning, a measured quantity $f$, such as the loss function minimized during training or a performance (e.g., precision) metric, is usually estimated as the average of the metric computed uniformly over the training or test set(s). Specifically, the estimate of $f$ over a dataset $D = \{x_i\}$ is computed as
>
> $$\widehat{f} = \frac{1}{|D|} \sum_{x_i \in D} f(x_i) \tag{1}$$
>
> Duplication biases this estimate because some $wx_i$ will appear multiple times. Specifically, we can equivalently transform $D$ as a multiset $X = \{(x_i, c_i)\}$ where $c_i \in \mathbb{N}+$ is the number of times that the sample $x_i$ is found in the dataset. Therefore, we can rewrite Equation 1 as
>
> $$\widehat{f} = (1-d) \underbrace{\frac{1}{|X|} \sum_{x_i \in X} f(x_i)}_{\text{unbiased estimate } \bar{f}} + d \underbrace{\frac{1}{|D|-|X|} \sum_{x_i \in X} (c_i - 1) f(x_i)}_{\text{duplication bias } \beta} \tag{2}$$
>
> where $d = \frac{|D|-|X|}{D} = \frac{\sum c_i - |X|}{D}$ is the *duplication factor* where $|X|$ is the number of unique $x_i$ in $X$. Thus $d$ is the proportion of the samples in the dataset that are duplicated ($c_i > 1$). By rewriting the above equation as $\widehat{f} = (1-d)\bar{f} + d\beta$ we see that the larger the duplicate factor $d$, the larger the effect of the duplication bias $\beta$.

[1]True data distribution is different from real-world data distribution since the former one should contain no duplicated samples while the latter one could.

From a machine learning perspective, the duplication bias in the training loss causes a model to overweight some training samples (the in-train duplicates). During testing, the duplication bias will skew the reported performance metric. Furthermore, we expect cross-set duplicates to artificially improve any metric taking advantage of the fact that multiple samples that are seen during training also appear in the test set, giving the illusion that the model generalizes, where in fact it memorized duplicates.

## 3 PUBLISHED DATASETS FOR MALWARE DETECTION

Our study aims at uncovering potential issues of stat-of-the-art ML-based malware detection approaches due to duplication bias. To this end, we first focus on investigating the presence of duplicates within commonly-used Android malware datasets in the described in the literature. Towards investigating the impact of sample duplication in ML-based Android malware detection, we are interested in knowing, in the first place, if sample duplication indeed exists in common Android Malware datasets. Our research question is thus:

**RQ1:** Does the duplication phenomenon, which has been revealed in big code modeling datasets, occur in Android malware datasets?

### 3.1 Study Datasets

Android Malware Detection and Analysis has received much attention for many years. The research community has collected and updated from various sources a variety of datasets that provide a ground truth for evaluating technical approaches to app malware analysis. Table 2 summarizes four representative ones[2] and provides some descriptive statistics about their size and diversity in terms of numbers of malware families that are represented. We then give a brief introduction to these four exemplar datasets.

Table 2. Statistics of Selected Android malware datasets.

| Dataset | #. Malware | Collecting Period | Release Date | #. Families |
|---|---|---|---|---|
| Genome | 1,260 | 2010 → 2011 | 2012 | 49 |
| Drebin | 5,560 | 2010 → 2012 | 2014 | $179^{\alpha}$ |
| AMD | 24,553 | 2010 → 2016 | 2017 | 71 |
| RmvDroid | 9,133 | 2014 → 2018 | 2019 | 56 |

$^{\alpha}$ adware is excluded from this dataset.

- **Genome.** The Genome project is a seminal work in the research of Android malware detection. As part of this project, Zhou et al. [70] publicly released a dataset of 1,260 malicious apps covering the majority of existing Android malware families (specifically, 49 families that were manually labeled by security analysts). The release dates of the app samples range from August 2010 to October 2011. Nowadays, Genome is considered to be obsolete, as most malware signatures have been well understood and malware writers are devising new techniques to hide malicious behavior (both from static checkers and dynamic monitoring).
- **Drebin.** To foster research on Android malware and to enable comparison among different detection approaches, Arp et al. [8] released the Drebin Android malware dataset in 2014,

[2]These datasets have been widely used by our community to evaluate the effectiveness of malware detection and classification approaches [60].

for which they built as part of their work on "explainable malware detection". The Drebin dataset contains 5,560 malicious apps that are collected between August 2010 and October 2012. This dataset also includes the samples from the Genome dataset. The 5,560 malicious apps are categorized by Drebin maintainers into 179 families. Unlike the families labeled in the Genome project, which are labeled mainly by practitioners, the malware samples in the Drebin dataset are labeled by the authors of the Drebin approach themselves based on the output of different anti-virus scanners. The authors took steps to manually unify the output of these anti-virus scanners.

- **AMD.** AMD is a carefully-labeled and well-studied Android malware dataset [61]. In addition to 24,553 samples assembled from 2010 to 2016, the dataset also includes a manually documented behavioral description of its malware samples. Based on the results of anti-virus scanners, the malware samples of this dataset are categorized into 135 variants within 71 malware families.
- **RmvDroid.** Released in 2019, RmvDroid [60] is the latest malware dataset that was released for complementing existing datasets, which are often outdated, unreliable, and lacking details of app metadata such as description, reviews, etc. The RmvDroid dataset contains 9,133 app samples that are associated with 56 malware families and were all caught after being exposed in the official Android market (i.e., Google Play).

Table 3 further enumerates some representative state-of-the-art approaches that have leveraged these datasets to train malware classification models. The last three columns of this table further illustrate the performance (i.e., precision, recall, and F1 score, respectively) achieved by those approaches. The fact that all the approaches have achieved over 97% F1 score (or 96% precision, 95% recall) suggests that these datasets selected in this work are representative and hence are suitable to fulfil our experiments.

Table 3. The performance achieved by some representative state-of-the-art approaches that have leveraged these datasets to train malware classification models.

| Dataset | Approach | Precision(%) | Recall(%) | F1 score(%) |
|---|---|---|---|---|
| Genome | Fan et al. [22] | 99.67 | 95.85 | 97.72 |
| Drebin | DANdroid [48] | 98.4 | 98.9 | 98.6 |
| AMD | Li et al. [35] | 99.22 | 99.16 | 99.19 |
| RmvDroid | Fan et al. [22] | 96.54 | 97.77 | 97.15 |

### 3.2 Duplication in Malware Datasets

An Android app is identified in the official market based on its unique package name, which prevents users from installing different versions of a given app on their device. However, since malicious code can be inserted during app updates, app versions may be considered as different samples within the real-world distribution of apps. Therefore, maintainers of datasets generally rely on hash values of APK files to ensure that sample APKs are unique.

*Cross-dataset APK duplication.* Although commonly-used datasets do not include duplicated APKs (the samples are often named by their SHA256 hash value), we note that the datasets are often redundant from one to another. This redundancy should be made clear to researchers who are interested in combining multiple datasets to fulfil their experiments [22, 65]. We hence provide in Table 4 statistics of cross-dataset duplications. We note that APK redundancy exists in all the considered malware datasets. The duplication rate varies from as small as 1% to as large as 97%, although those malware datasets are all collected via different methods. Even though Drebin authors

have claimed that it included all the Genome samples, due to some outlier cases, we cannot observe a 100% duplication rate for these two datasets.

Table 4. APK duplication between Selected Android malware datasets. The duplication rate is calculated via the following formula: $\frac{|Dataset1 \cap Dataset2|}{min(|Dataset1|,|Dataset2|)}$.

| Dataset1 | Dataset2 | #. Cross | Duplication Rate | Dataset1 | Dataset2 | #. Cross | Duplication Rate |
|---|---|---|---|---|---|---|---|
| AMD | RmvDroid | 2,618 | 28.67% | AMD | Genome | 365 | 28.97% |
| Drebin | Genome | 1,229 | 97.54% | Drebin | RmvDroid | 40 | 0.72% |
| AMD | Drebin | 559 | 10.05% | Genome | RmvDroid | 26 | 2.06% |

*Within-dataset duplication.* While researchers appear to ensure that datasets are not duplicated at the APK level, we note that the relevant in-APK components may be duplicated. For example, including several repackaged versions of a given app, where only layout and image changes are applied, will lead to a duplicated dataset of code (Dex) clones. Similarly, looking at lower-level details that constitute the samples for learning, one might discover new duplications that prevent to faithfully model the distribution of data that practitioners have to deal with. Fig. 1 summarizes the statistics of duplication for dex code, opcode sequence, and API calls in all the four selected datasets.

We note that Dex code duplication concerns between 2.6% of samples (the RmvDroid dataset) and 40% of samples (the Drebin dataset). For the case of the Drebin dataset, this means that for 40% of its app samples, there exists at least one other sample in the dataset that shares the same DEX file with them. Although the percentages are substantial, they are far less than the percentages of duplication for Opcode sequences (minimum of 30%) and API calls (minimum of 40%). Distribution of the number of samples in each duplication are further provided in Fig. 2 to provide more insights. The median values of all the distributions[3] shows that at least half of the duplications happen on only two samples, indicating the selected datasets are quite diverse despite the existence of sample duplication. The fact that the maximum values of all the distributions are always less than five also backs up this indication.

Table 5. Malware Family intersection between selected Android malware datasets. The intersection rate is calculated via the following formula: $\frac{|Dataset1 \cap Dataset2|}{min(|Dataset1|,|Dataset2|)}$.

| Dataset1 | Dataset2 | #. Same Families | Intersection Rate | Dataset1 | Dataset2 | #. Same Families | Intersection Rate |
|---|---|---|---|---|---|---|---|
| AMD | RmvDroid | 12 | 21.43% | AMD | Genome | 5 | 10.87% |
| Drebin | Genome | 18 | 39.13% | Drebin | RmvDroid | 10 | 17.86% |
| AMD | Drebin | 21 | 29.58% | Genome | RmvDroid | 5 | 10.87% |

### 3.3 Family representation in datasets (a.k.a. family duplication)

During our analyses, we have found that although datasets include samples from a large variety of malware families, the intersections of families between the selected datasets are relatively small. As summarized in Table 5, the intersection rate ranges from 10.87% to 39.13%. The actual distribution of the size of malware families is often highly imbalanced. For instance, in the Drebin dataset, within the 179 families that are enumerated, some are represented with a single APK sample, while other families include hundreds of samples, as illustrated in Fig. 3. This imbalance, if ignored, may introduce biases to malware clustering approaches and hence the performance of ML-based malware classifications. Indeed, as argued by Yu et al. [67], in practice, samples in a dataset may have

[3]The only difference is *Dex in the RmvDroid dataset, for which the median value is also quite close to 2.*

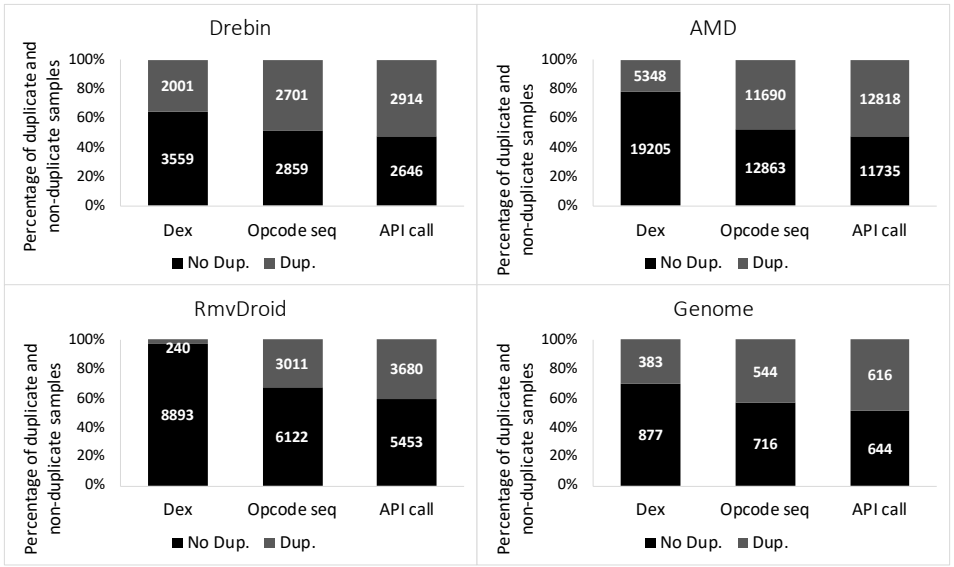


Fig. 1. Within-dataset sample duplication in the selected malware datasets.

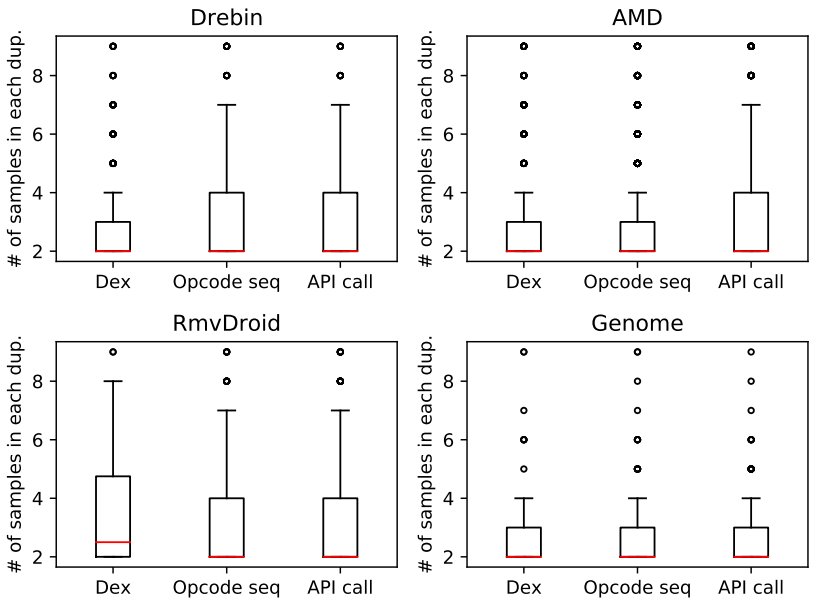


Fig. 2. Distribution of the number of samples in each duplication (Samples without duplication are ignored).

different importance when conducting clustering-based approaches. Therefore, it is important to properly adjust the sample distributions (or weights) when clustering a dataset. Unfortunately, the same phenomenon occurs for RmvDroid, AMD, and Genome datasets. For the sake of characterizing the imbalance among various families, we counted the malware samples from the top 10 families of

each dataset and made a comparison with the whole dataset, as shown in Fig 4. For all the malware datasets, the top-10 family samples account for over 70% of the total samples. As shown in Table 6, samples in some malware families may have been highly duplicated (could be over 90%) as Android malware developers tend to create new malware by repackaging existing ones.

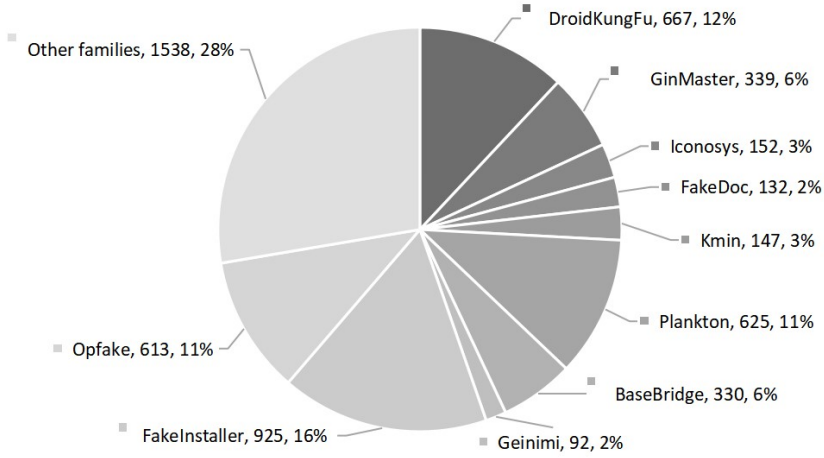


Fig. 3. Family distribution of the Drebin dataset.

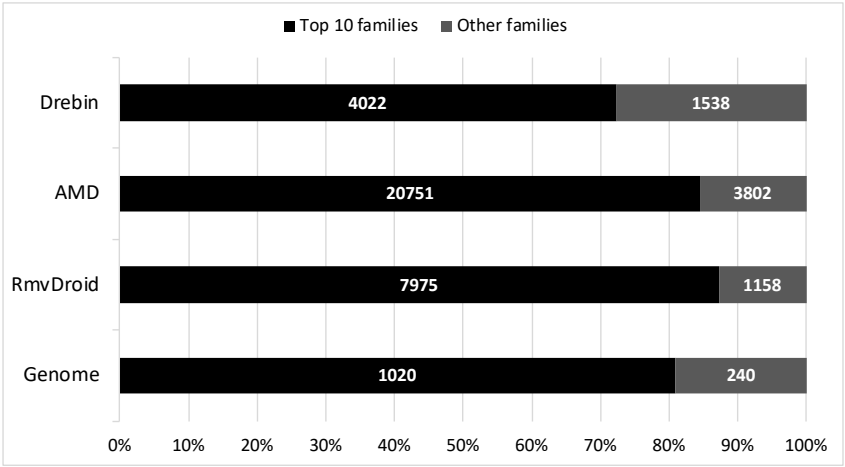


Fig. 4. Imbalanced distribution of malware families in the selected four datasets.

Table 6. Sample duplication in the top-10 malware families of the selected Android datasets.

| Drebin | | | | AMD | | | | RmvDroid | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Family** | **Dex** | **Opcode seq** | **API call** | **Family** | **Dex** | **Opcode seq** | **API call** | **Family** | **Dex** | **Opcode seq** | **API call** |
| FakeInstaller | 82.27% | 84.86% | 85.62% | Airpush | 1.05% | 36.23% | 42.97% | Airpush | 2.37% | 43.28% | 48.93% |
| DroidKungFu | 18% | 34.93% | 44.98% | Dowgin | 4.73% | 22.62% | 26.2% | Mecor | 0 | 88.83% | 91.48% |
| Plankton | 0.32% | 18.72% | 22.88% | FakeInst | 82.7% | 85.29% | 88.38% | Plankton | 0 | 12.59% | 19.88% |
| Opfake | 80.91% | 86.79% | 87.11% | Mecor | 0.27% | 97.58% | 97.91% | Adwo | 0.28% | 10.64% | 39.5% |
| GinMaster | 2.95% | 3.24% | 3.24% | Youmi | 0.69% | 16.94% | 23.94% | Youmi | 0 | 11.35% | 21.31% |
| BaseBridge | 60.91% | 74.55% | 77.58% | Fusob | 89.83% | 89.83% | 94.25% | Mobidash | 30.97% | 78.13% | 82.39% |
| Iconosys | 0 | 41.45% | 50.66% | Kuguo | 1.58% | 21.85% | 25.77% | Kuguo | 0.26% | 3.6% | 4.63% |
| Kmin | 42.18% | 67.35% | 70.07% | BankBot | 58.91% | 80.87% | 83.7% | Gappusin | 0.65% | 6.1% | 18.74% |
| FakeDoc | 68.18% | 69.7% | 69.7% | Jisut | 72.71% | 85.35% | 91.21% | Viser | 0.34% | 9.52% | 13.27% |
| Geinimi | 15.22% | 21.74% | 26.09% | DroidKungFu | 18.68% | 33.88% | 45.05% | Dowgin | 2.38% | 7.48% | 9.18% |

**RQ-1** Answer

Duplication is commonplace in malware datasets used in the published literature. It occurs for samples at a different artefact level when leveraged for machine learning based malware detection. Duplication may concern up to 40% of Dex code samples in datasets that are widely used for experimental validation of detection techniques, or up to 97% in certain malware families, which may significantly bias evaluation results.

## 4 STUDY DESIGN

We now present the detailed design of our empirical study, including the research questions we aim to answer (Subsection 4.1), the machine learning algorithms we will leverage in this work (Subsection 4.2), the features we plan to extract from Android apps (Subsection 4.3), and the evaluation metrics we will leverage to categorize the performance of machine learning classifications (Subsection 4.4).

### 4.1 Research Questions

Machine learning techniques for malware detection have been largely assessed using common datasets that include duplicated artefacts. We conduct several experiments to assess for any potential duplication bias, specifically the extent of sample duplication impact on the performance of state of the art ML-based Android malware detectors. To that end, we propose three additional refined research questions that explore cases of both supervised and unsupervised learning approaches.

- **RQ2:** What is the impact of malware sample duplication on supervised learning for building Android malware detectors?
- **RQ3:** Is the impact of sample duplication influenced by a specific underlying supervised learning algorithm?
- **RQ4:** Are unsupervised learning models impacted by sample duplication bias in a similar way to supervised learning models?

### 4.2 Machine Learning Algorithms

We now present the machine learning algorithms that are leveraged in this work to implement Android malware predictors. Using several state-of-the-art malware detection approaches described in the literature, we train binary classification models to predict malicious content of sample apps. Specifically, we have focused on the following four algorithms. We first leverage SVM to answer RQ2 (cf. Section 5.1) and then leverage all the four algorithms to empirically compare the impact of duplication bias on different machine learning algorithms (cf. Section 5.2).

- **Support Vector Machine (SVM)** is a widely used supervised machine learning algorithm that attempts to perform binary classifications by finding the hyperplane that effectively separates data points associated with two classes [14]. In malware detection tools, SVM has been one of the key long term algorithms that have been explored. Even state of the art approaches such as Drebin have achieved high performance by relying on SVM [8].
- **Decision Trees (DT)** infers classification rules from a set of unordered and irregular cases. It performs classification using branch structure, using a tree as a form of expression. According to the survey of Safavian et al. [55], decision trees have been successfully used in many diverse areas. Aung et al. [10] have used decision trees to classify Android applications as malicious or benign by focusing on their permission features.
- The **K-Nearest Neighbor (KNN)** algorithm identifies the closest K instances (from the training dataset) based on a certain distance metric and from which it picks up the most common class tag among to form the prediction result. The study of Firdausi et al. [23] shows that simple machine learning algorithms such as KNN can be used to detect malicious applications effectively and efficiently.
- **Random Forest (RF)** is a classification algorithm that builds multiple decision trees from which the final output is converged by voting the results yielded by individual trees. Several tools for Android app analysis have used RF for malware classification. For example, Alam et al. [3] have largely relied on RF in their study. Sanz et al. [56] have found that RF produces the best classifier among all the algorithms including Sequential Minimal Optimization (SMO) [1], KNN, etc.

### 4.3 Feature engineering

Machine learning-based classification relies on data to learn what characteristics could suggest that a given sample likely belongs to a given class. For example, in malware detection, the learning algorithms must be "told" what characteristics are associated with each malicious or benign sample in the dataset. Such characteristics are known as the feature set and implemented as a feature vector. Arp et al. [8] proposed one of the most comprehensive feature sets for Android malware detection, which has proved effective in the state of the art Drebin approach. They focused on features that can be extracted with a lightweight static analysis approach in order to scale their tool to thousands of samples. Such features are based on the Manifest file in the apk (i.e., *AndroidManifest.xml*) as well as the disassembled DEX code. In the end, eight (8) aspects are considered to produce 8 sets of strings:

- **S1: Hardware components.** Malicious behavior in Android malware often involves access to specific hardware components such as the camera or the GPS. Related features statically extracted based on access requests made can be derived from the manifest file.
- **S2: Requested permissions.** Malicious apps tend to request specific permissions more frequently, such as SEND_SMS, CAMERA, READ_CONTACTS than that of benign apps [56]. Hence permission requests from the Manifest file could be a good indicator to differentiate malware from goodware.
- **S3: App components.** Activities, services, content providers and broadcast receivers that are declared in the manifest file are the four types of existing components that define different specific interfaces to the system. Some components may be statistically more leveraged by malicious apps than by benign apps.
- **S4: Filtered intents.** An Intent is a messaging object used to exchange data between app components. Malware may use it to listen to particular intents to achieve malicious purposes, e.g., malware could hijack the SMS_RECEIVED system intent to listen to users' SMS messages.

- **S5: Restricted API calls.** A typical case of identifying malicious behavior is the use of restricted API calls without requesting required permissions, i.e., the application has a high probability of gaining external permissions through an exploit of privilege escalation vulnerability. Such restricted critical calls can be detected from the disassembled code.
- **S6: Used permissions.** After harvesting the restricted API calls (as shown in S5), one can calculate the actual permissions required by the app accessed those APIs. This permission set might be different from the one explicitly declared by app developers in the manifest (cf. S2).
- **S7: Suspicious API calls.** Certain API calls may cause exposure to sensitive data or resources and are often exploited by Android malware. API calls for sensitive data access, network communication, sending and receiving SMS messages, executing external commands and frequently used for obfuscation are gathered.
- **S8: Network addresses.** In many cases, malicious apps eventually need to fetch external data (e.g., download dynamically loaded code) or to leak data outside the app. These activities require network communication with specific hosts. Thus, IP addresses, hostnames and URLs gathered from the disassembled code can be characterized for malicious behavior prediction.

The aforementioned feature set covers a wide range of characteristics of Android apps and has been adopted widely by the research community [2, 20, 27, 47]. Our experiments in this work directly leverage this comprehensive feature set to build relevant classifiers that match the typical performance recorded in the literature. Consecutively, our study focuses on the impact of sample duplication bias on these classifiers. We note that the eventual size of the feature set in each experiment is dependent on the app dataset selected for training.

### 4.4 Evaluation Metrics

For a binary classification problem, the ML model ultimately needs to predict whether a given sample belongs to one of two classes (i.e., generally is positive or negative). In our case, we consider the classes *malware* and *goodware*. A confusion matrix is often used to establish the performance of a classification model, based on four measurements:

(1) True Positive ($TP$), i.e., the number of malware samples are flagged as malicious by the ML model,
(2) False Negative ($FN$), i.e., the number of goodware samples are flagged as benign by the ML model,
(3) True Negative ($TN$), i.e., the number of malware samples are flagged as benign by the ML model, and
(4) False Positive ($FP$), i.e., the number of goodware samples that are flagged as malicious by the ML model.

Based on these enumerations, we can compute the following three metrics (cf. *Precison*, *Recall*, and $F1$ *Score*) that are commonly taken as indicators for evaluating ML-based approaches, and are defined as follows.

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1\ score = \frac{2 * Precision * Recall}{Precision + Recall}$$

## 5 EXPERIMENTAL RESULTS

In this section, we detail the experimental results we observed for answering our four research questions from Section 4.1. The four research questions cover both 10-fold cross-validation (RQ2) and holdout experiments (RQ 2-3), supervised learning (RQ 2-3) and unsupervised learning (RQ4) approaches, as well as a comparison of different machine learning algorithms (RQ3).

### 5.1 RQ2: Impact of Duplication Bias on Malware Classifiers

Our experiments to assess the impact of duplication bias in malware datasets follow the ML-based approach for malware detection proposed by Arp et al. [8] (i.e., we train the machine learning model through the famous SVM algorithm). We leverage the common datasets used in the Android malware detection literature (namely Drebin, AMD and RmvDroid) and investigate the impact of all the three levels of duplications (i.e., DEX file, Opcode Sequence, and API Call) in machine learning based malware detection. To better characterize the impact of duplication bias, following the general practice of state-of-the-art works [5, 52], we answer this research question in two experimental settings: (1) 10-fold cross-validation (also known as in-the-lab experiments), which is a widely used statistical method for estimating the capability of machine learning models. (2) in-the-wild experiments, which attempt to evaluate the capability of machine learning models in a real-world setting, i.e., training the model based on a known dataset and use the model to predict unknown dataset. Below we now describe these two experiments in detail.

*5.1.1 10-fold Cross-validation.* In 10-fold cross-validation, the dataset is randomly divided into 10 equal-size sample sets. Among the 10 subsets, one of them is retained as the test set for validating the performance of the machine learning model, which is then trained on the remaining 9 subsets. This process is then repeated 10 times with each of the subsets used exactly once as the test set. The performance measurements of these 10 validations are then averaged to compute the final performance metric of the classification approach.

**Experimental Setup.** Given a malware dataset and a duplicated sample type, we set up two experiments: one without sample duplication while another one with sample duplication. These two experiments form a controlled group for evaluating the difference brought by sample duplication to the machine learning-based classifications. To better distinguish the settings, we set up six experiments, two for each duplication type, for each malware dataset, as detailed below.

- **E1/E2: Without/With DEX Duplication.** These two settings evaluate the performance of machine learning approaches w.r.t. DEX duplication. For E1, i.e., without DEX duplication, the training set is formed by taking into account all the non-duplicated samples from the original dataset. For E2, we randomly select the same number of samples (as that of E1) from the original dataset to form the training set. Since the original dataset contains duplicated samples, the randomly selected subset will likely contain duplicated samples as well. Indeed, the chance to randomly select a set of $|E1|$ (i.e., 3,559) apps from the original dataset (i.e., 5,560 apps) that the selection is identical to E1 is low.
- **E3/E4: Without/With Opcode Sequence Duplication.** Similar to **E1/E2** except that *opcode sequence duplication* is used instead of *dex file*.
- **E5/E6: Without/With API Call Duplication.** Similar to **E1/E2** except that *API call duplication* is used instead of *dex file*.

Since we are interested in conducting binary classifications (i.e., malware or goodware) and two of the three considered datasets do not come with benign apps (the datasets contain malware only), we relied on the AndroZoo repository [6, 37, 43] to collect benign samples to train the machine learning model. AndroZoo is a growing collection of Android apps collected from several sources,

Table 7. SVM-based Android malware detection via 10-fold cross validation. To support binary classification, we randomly select the same number of benign apps like that of malware to form the training set (i.e., #. malware *2).

| Dataset | Type | Setting | Training Set | # Features | # Duplicated Vectors (Ratio) | Malware | | | Goodware | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Precision(%) | Recall(%) | F1 score(%) | Precision(%) | Recall(%) | F1 score(%) |
| Drebin | Dex | E1 (Without Duplication) | 3,559*2 | 48,026 | 949 (26.66%) | 96.56 | 94.91 | 95.72 | 95.03 | 96.62 | 95.81 |
| | | E2 (With Duplication) | 3,559*2 | 43711 | 2,019 (56.72%) | 97.82 | 95.84 | 96.81 | 95.96 | 97.85 | 96.89 |
| | Opcode seq | E3 (Without Duplication) | 2,859*2 | 44,494 | 455 (15.91%) | 96.2 | 94.58 | 95.38 | 94.71 | 96.28 | 95.48 |
| | | E4 (With Duplication) | 2,859*2 | 40,305 | 1,338 (46.8%) | 97.21 | 95.24 | 96.21 | 95.37 | 97.28 | 96.31 |
| | API call | E5 (Without Duplication) | 2,646*2 | 39,388 | 316 (11.94%) | 96.18 | 93.91 | 95.02 | 94.11 | 96.28 | 95.17 |
| | | E6 (With Duplication) | 2,646*2 | 35,117 | 1,162 (43.93%) | 97.12 | 95.1 | 96.09 | 95.24 | 97.19 | 96.2 |
| AMD | Dex | E1 (Without Duplication) | 19,205*2 | 185,011 | 3,299 (17.18%) | 98.05 | 95.64 | 96.78 | 95.87 | 98.04 | 96.9 |
| | | E2 (With Duplication) | 19,205*2 | 172,507 | 4,730 (24.63%) | 98.21 | 95.92 | 96.99 | 97.19 | 97.06 | 97.06 |
| | Opcode seq | E3 (Without Duplication) | 12,863*2 | 143,889 | 1,545 (12.01%) | 97.78 | 95.77 | 96.71 | 96 | 97.79 | 96.84 |
| | | E4 (With Duplication) | 12,863*2 | 122,438 | 4,629 (35.99%) | 98.14 | 96.13 | 97.07 | 96.34 | 98.14 | 97.19 |
| | API call | E5 (Without Duplication) | 11,735*2 | 135,485 | 1,015 (8.65%) | 97.48 | 94.93 | 96.09 | 95.32 | 97.5 | 96.32 |
| | | E6 (With Duplication) | 11,735*2 | 114,330 | 3,895 (33.19%) | 97.85 | 96.13 | 96.94 | 96.31 | 97.86 | 97.04 |
| RmvDroid | Dex | E1 (Without Duplication) | 8,893*2 | 100,998 | 1,398 (15.72%) | 97.79 | 98.45 | 98.11 | 98.44 | 97.73 | 98.07 |
| | | E2 (With Duplication) | 8,893*2 | 100,100 | 1,561 (17.55%) | 97.81 | 98.49 | 98.14 | 98.48 | 97.75 | 98.11 |
| | Opcode seq | E3 (Without Duplication) | 6,122*2 | 77,693 | 432 (7.06%) | 96.79 | 98.25 | 97.5 | 98.23 | 96.67 | 97.42 |
| | | E4 (With Duplication) | 6,122*2 | 71,079 | 1,011 (16.51%) | 97.15 | 98.4 | 97.75 | 97.77 | 97.72 | 97.72 |
| | API call | E5 (Without Duplication) | 5,453*2 | 73,467 | 273 (5.01%) | 96.35 | 98.06 | 97.18 | 98.02 | 96.23 | 97.1 |
| | | E6 (With Duplication) | 5,453*2 | 62,587 | 1,030 (18.89%) | 97.3 | 98.33 | 97.89 | 98.32 | 97.23 | 97.75 |

including the official Google Play app market. It currently contains over 10 million Android APKs. Each of them has been scanned by over 70 anti-virus products hosted on VirusTotal[4]. We consider an app to be benign as long as none of the anti-virus products (hosted on VirusTotal) flags it as malware. Specifically, for each experimental setting, we randomly select the same number of goodware (as that of malware) to form the training set, as unbalanced training sets may introduce biases to machine learning based classifications. Furthermore, to avoid potential biases introduced by our random sampling, we ensure that there are no repackaged app pairs between the selected malware and goodware samples (i.e., do not share the same package names). We also conduct each experiment setting 10 times and report the average performance as the output. These settings apply to all the experimental results reported in this paper.

**Result.** Table 7 summarizes the 10-fold cross-validation results. Following the experimental setup, for each dataset, we perform six different experiments (i.e., E1 → E6): two experiments for a given sample duplication type. As indicated in the fourth column, different duplication types will result in a different number of samples for training, which subsequently leads to a different number of features (as shown in the fifth column). Generally, the more training samples considered, the larger the feature sets extracted.

The last six columns in Table 7 further illustrate the classification results for predicting both malware and goodware of the SVM-based malware detection approach. Based on these results, we can observe the following interesting findings:

**Finding-2.1:** When predicting goodware, no matter which metric is considered, or which duplication type is concerned, the performance achieved via training datasets containing duplicated samples is always higher than that of datasets without duplicated samples.

**Finding-2.2:** Regarding malware prediction, the results are more or less similar to that of goodware.

These two findings suggest that machine learning models tend to achieve higher performance if there are overlaps between the training sets and test sets. Indeed, when performing 10-fold cross-validations, duplicated samples will likely be divided into both training and test sets. This verdict further implies that machine learning approaches, in practice, should be trained on datasets that are as comprehensive and representative as possible. The more representative samples we can include in the training set, the more likely there will be similar samples in the test set, and thereby

[4]https://www.virustotal.com/gui/

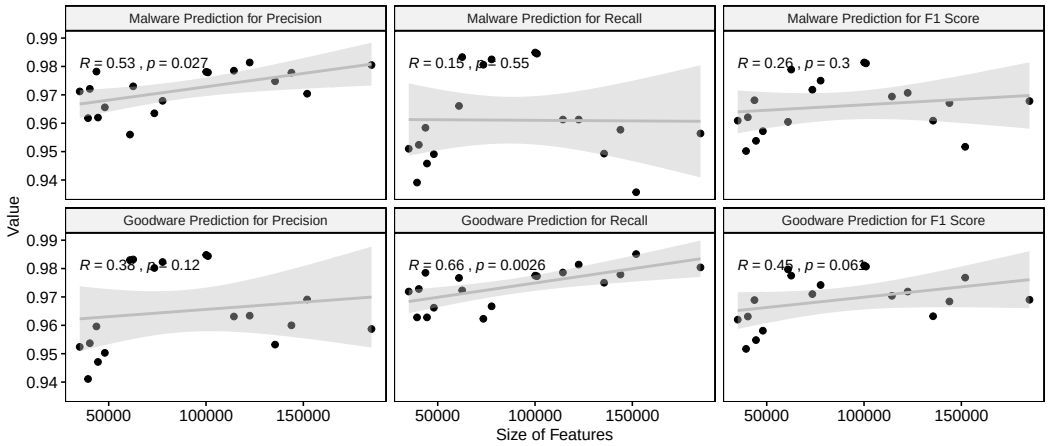


Fig. 5. The correlation between feature size and the classification results (i.e., Precision, Recall, and F1 score). Each dot represents one experiment.

the higher the performance machine learning approaches could achieve. Nevertheless, we argue that researchers, when reporting the performance of their machine learning approaches, should make it clear if sample duplication exists in their dataset.

Interestingly, as demonstrated in Table 7, the performance difference between a controlled experimental pair E1 and E2, no matter which metric is concerned, is always smaller than 1.32%. This evidence suggests that the actual impact brought by sample duplication is insignificant. In other words, the validity of state-of-the-art malware detection approaches may not be severely threatened due to sample duplication of their training dataset. Nevertheless, we argue that duplicates (1) should still be removed to avoid unnecessary biases in machine learning-based classifications, or (2) be kept if a clear and convincing argument can be given.

Fig. 5 further illustrates the correlation between feature sizes and the performance of the machine learning approach. The fact that $R - value$ is positive for all the cases shows that the performance of the 10-fold cross-validation is generally aligned with the size of features considered. Nonetheless, some of the positive correlation is quite weak and yet not significant, given a significance level of $\alpha = 0.001$[5]. This finding further confirms that the impact brought by sample duplication might be marginal to the performance of machine learning approaches.

*5.1.2 In-the-wild Experiments.* As advocated by Allix et al. [5], when conducting machine learning-based Android malware detection, in-the-lab experiments, such as using 10-fold cross-validation, may not be reliable to justify the performance of the machine learning models. There is also a need to validate the performance of the machine learning models in a real-world setting, i.e., in-the-wild experiments such as training on a dataset while testing on another dataset. In our second research question, we re-evaluate the impact of sample duplication for machine learning-based malware detection approaches through a so-called in-the-wild experimental setting.

**Experimental Setup.** As illustrated in Fig. 6, given a malware dataset, we create two subsets, namely $ND$ and $D$. We put all samples one by one that do not introduce duplication into $ND$, and those that involve duplication into $D$. As a result, all the samples in $ND$ do not contain duplicated samples while all the samples in $D$ have duplicated versions presented in $ND$. We then randomly

[5]There is one chance in a thousand that the difference between the datasets is due to a coincidence.

select a small set of samples, $Y$, in $ND$ to form the test set[6] and prepare the training set in two settings, noted as S1 and S2 in Fig. 6).

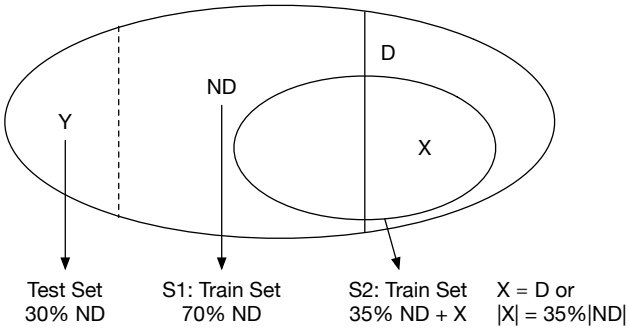


Fig. 6. The experiment design of ML-based Malware Detection. $X$ stands for the set of samples selected from $|D|$ while $Y$ stands for the set of apps selected for fulfilling the test set, which is kept the same for both $S1$ and $S2$.

In this experiment, we set $Y$ as 30% of the samples in $ND$ (i.e., 30% of samples with no duplication). Consequently, the training malware set contains $|70\%ND|$ samples (i.e., either $ND-Y$ or $35\%ND+X$, where $X$ can be calculated via the following formula.

$$X = \begin{cases} D, & |D| <= |35\%ND| \\ K, & K \in D \text{ and } |K| = |35\%ND| \end{cases}$$

In rare cases, if $|D| < 35\%|ND|$, meaning that we cannot select the same number of apps from the set of $D$ to form the training set with duplication, we simply select all the apps in the $D$ set to form the training malware set. Recall that we aim at conducting binary classification in this work. Therefore, we need to add goodware to the training set as well. To this end, we randomly select the same number of goodware (i.e., $|70\%ND|$) from Google Play to form the final training set.

Furthermore, instead of arbitrarily stipulating the values of hyper-parameters for the machine learning model, we leverage the *grid search* technique to automatically find suitable values for those hyper-parameters. *Grid search* is a popular pre-process step that explores all the possible parameter combinations to pinpoint an optimal combination for the model.

**Result.** Table 8 summarizes our experimental results for the experiments conducted for answering RQ2, for which SVM binary classification is applied to all the three representative datasets. When a different duplication type is involved, the total number of non-duplicated and duplicated samples will be different. In all datasets, Dex duplication yields the largest number of non-duplicated samples, followed by Opcode Sequence and API Call, respectively. The different size of non-duplicated samples subsequently causes different training and test sets, which further yields different numbers of features for the machine learning classification. The number of features is dependent on the selected training dataset. Different apps may contribute to different features, although the same extraction strategy is applied.

- **Finding-2.3:** Unlike the results we obtained previously, i.e., via 10-fold cross-validation, in this experiment, as highlighted in the table, training set with sample duplications often achieves a higher precision yet lower recall for predicting malware than such settings without

[6]It is worth noting that the test set will have no impact on the performance of machine learning models. We kept the test set duplication-free to avoid potential biases.

Table 8. Experimental results of SVM-based malware classification. $|ND|$ means the number of non-duplicated samples (after excluding duplicated ones). To enable binary classification, the train and test sets are fitted with randomly selected goodware.

| Dataset | Type | Setting | $\|ND\|$ | Training Set | Test Set | # Features | # Duplicated Vectors (Ratio) | Malware | | | Goodware | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Precision(%) | Recall(%) | F1 score(%) | Precision(%) | Recall(%) | F1 score(%) |
| Drebin | Dex | E1 (Without Duplication) | 3,559 | 2,491*2 | 1,068*2 | 34,799 | 581 (23.32%) | 93.4 | 96.81 | 95.08 | 96.69 | 93.16 | 94.89 |
| | | E2 (With Duplication) | 3,559 | 2,491*2 | 1,068*2 | 31,299 | 1,341 (53.85%) | 94.34 | 95.31 | 94.82 | 95.27 | 94.27 | 94.77 |
| | Opcode seq | E3 (Without Duplication) | 2,859 | 2,001*2 | 858*2 | 33,398 | 255 (12.74%) | 91.37 | 96.38 | 93.81 | 96.17 | 90.9 | 93.46 |
| | | E4 (With Duplication) | 2,859 | 2,001*2 | 858*2 | 29,843 | 864 (43.17%) | 92.92 | 94.54 | 93.73 | 94.44 | 92.8 | 93.61 |
| | API call | E5 (Without Duplication) | 2,646 | 1,852*2 | 794*2 | 30,289 | 206 (11.12%) | 91.89 | 95.71 | 93.76 | 95.53 | 91.55 | 93.5 |
| | | E6 (With Duplication) | 2,646 | 1,852*2 | 794*2 | 26,942 | 756 (40.81%) | 92.76 | 94.74 | 93.74 | 94.63 | 92.6 | 93.6 |
| AMD | Dex | E1 (Without Duplication) | 19,205 | 13,444*2 | 5,761*2 | 135,015 | 2,745 (20.42%) | 97.6 | 99.55 | 98.56 | 99.54 | 97.55 | 98.54 |
| | | E2 (With Duplication) | 19,205 | 13,444*2 | 5,761*2 | 111,349 | 5,838 (43.42%) | 97.91 | 99.33 | 98.61 | 99.32 | 97.88 | 98.59 |
| | Opcode seq | E3 (Without Duplication) | 12,863 | 9,004*2 | 3,859*2 | 107,270 | 909 (10.1%) | 97.09 | 99.43 | 98.25 | 99.42 | 97.02 | 98.2 |
| | | E4 (With Duplication) | 12,863 | 9,004*2 | 3,859*2 | 90,748 | 3,058 (33.96%) | 97.2 | 99.17 | 98.17 | 99.16 | 97.13 | 98.13 |
| | API call | E5 (Without Duplication) | 11,735 | 8,215*2 | 3,520*2 | 101,418 | 585 (7.12%) | 97.3 | 99.32 | 098.3 | 99.3 | 97.24 | 98.26 |
| | | E6 (With Duplication) | 11,735 | 8,215*2 | 3,520*2 | 85,655 | 2,569 (31.27%) | 97.2 | 99.41 | 98.29 | 99.39 | 97.14 | 98.25 |
| RmvDroid | Dex | E1 (Without Duplication) | 8,893 | 6,225*2 | 2,668*2 | 73,911 | 898 (14.43%) | 96.85 | 98.99 | 97.9 | 98.96 | 96.78 | 97.86 |
| | | E2 (With Duplication) | 8,893 | 3,353*2 | 2,668*2 | 73,211 | 1,066 (31.79%) | 96.86 | 98.97 | 97.9 | 98.95 | 96.79 | 97.86 |
| | Opcode seq | E3 (Without Duplication) | 6,122 | 4,285*2 | 1,837*2 | 58,773 | 247 (5.76%) | 95.1 | 99.4 | 97.2 | 99.37 | 94.88 | 97.07 |
| | | E4 (With Duplication) | 6,122 | 4,285*2 | 1,837*2 | 50,276 | 869 (20.28%) | 95.73 | 98.85 | 97.26 | 98.81 | 95.59 | 97.17 |
| | API call | E5 (Without Duplication) | 5,453 | 3,817*2 | 1,636*2 | 52,695 | 160 (4.19%) | 95.85 | 98.84 | 97.32 | 98.8 | 95.72 | 97.24 |
| | | E6 (With Duplication) | 5,453 | 3,817*2 | 1,636*2 | 44,554 | 680 (17.82%) | 96.15 | 98.55 | 97.34 | 98.51 | 96.06 | 97.27 |

duplication involved in the training sets. This finding implies that, with duplication samples in the training set, the classifier is more conservative and less likely to flag a given app as malware, resulting in a lower false-positive rate and hence a higher false-negative rate. In opposite, without sample duplication, the classifiers are able to achieve better recalls. This could be explained by the fact that more diverse samples of malware[7] are learned by the classifiers, making them more knowledgeable to pinpoint unknown ones.

- **Finding-2.4:** Different datasets will yield different classification results, which are further aligned with the size of the training dataset, which is similar to the result obtained via 10-fold cross-validation. In our experiment, AMD has the largest training dataset and achieves the best performance in terms of distinguishing malware from benign ones. The impact of sample duplication on machine learning-based malware detection can vary from dataset to dataset. For example, the performance yielded by the AMD dataset is more stable than that of the Drebin dataset, which yields a larger range of vibration of results. This observation further suggests that the dataset quality is very important for machine learning based Android malware detection.

**RQ-2** Answer

When performing 10-fold cross-validation for Android malware detection, sample duplication has a positive impact on the performance of classification results. We hence advocate that practitioners and researchers should pay attention to sample duplication when conducting ML-based classification results. Nevertheless, the fact that the performance difference is quite small suggests that the impact brought by sample duplication to machine learning approaches might be marginal.

When performing in-the-wild experiments for predicting malware, sample duplication also impacts the performance of ML-based classification results. The fact that the performance of ML models varies from dataset to dataset further suggests that the dataset quality is important for ML-based Android malware detection approaches. Nonetheless, similar to that of 10-fold cross-validation, the impact of sample duplication on the machine learning models is marginal. We argue that duplicates (1) should still be removed to avoid unnecessary biases in machine learning-based classifications, or (2) be kept if a clear and convincing argument can be given.

[7] Given fixed number of malware, the more duplicated malware samples included, the less diverse and representative the malware set will be.

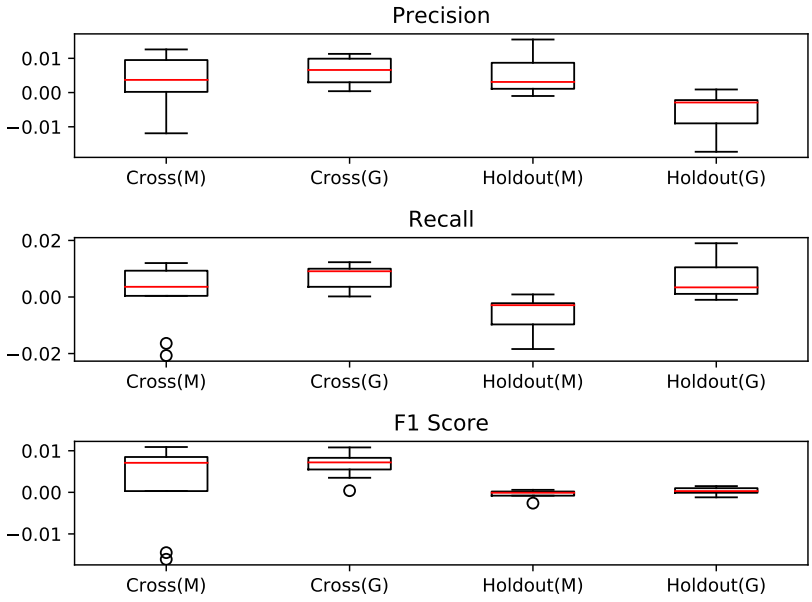


Fig. 7. Distribution of performance differences between 10-fold cross validation and the holdout experiments.

## 5.2 RQ3: Impact of Duplication Bias on Different ML Algorithms

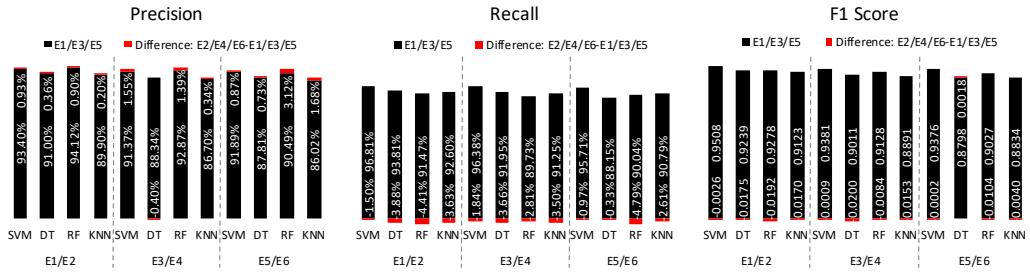


Fig. 8. Experiment results with different ML algorithms on Drebin dataset.

As elaborated in Section 4.2, there are various machine learning algorithms recurrently leveraged by researchers for detecting Android malware. Specifically, we further consider three supervised machine learning algorithms, namely Decision Tree (DT), RandomForest (RF), and K-Nearest Neighbors (KNN).

**Experimental Setup.** The experimental setup for answering this research question is the same as the one leveraged for answering RQ2 (e.g., $E1 \rightarrow E6$) except that the machine learning algorithms are now altered to DT, RF, and KNN, respectively. Similar to the experiments of SVM-based classification, we also leverage *grid search* to automatically stipulate hyper-parameter values for the newly selected machine learning algorithms.

**Results.** Fig. 8, Fig. 9, and Fig. 10 illustrate these experimental results. For each dataset, the precision, recall, and F1 score values are presented, and for each experimental setting, four machine

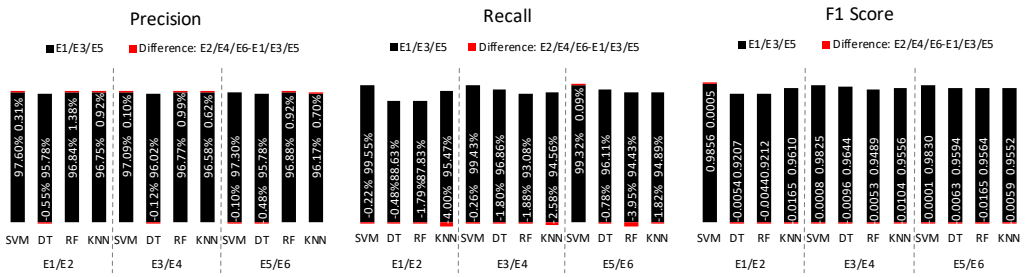


Fig. 9. Experiment results with different ML algorithms on AMD dataset.

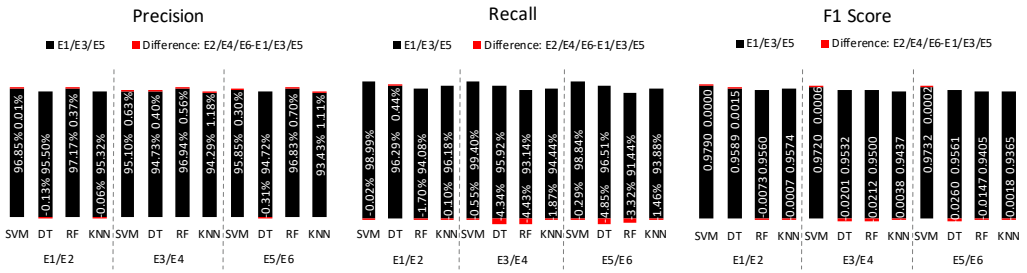


Fig. 10. Experiment results with different ML algorithms on RmvDroid dataset.

learning algorithm results are comparatively illustrated. Based on these results, we summarize the following findings.

- **Finding-3.1:** When different evaluation metrics are considered, the performance of different machine learning algorithms is also different, although they are applied to the same dataset. Indeed, for precision, KNN yields the worst performance for Drebin and RmvDroid datasets, while DT yields the worst performance for the AMD dataset. For recall, RF yields the worst performance for all the three considered datasets. For F1 scores, SVM always achieves the best performance compared to the other ML algorithms.
- **Finding-3.2:** Sample duplication has diverse impacts on these different machine learning algorithms. As shown in Fig. 8-10, in the 36 controlled experimental groups – four machine learning algorithms are evaluated in each pair, e.g., SVM, DT, RF, and KNN in E1/E2, Precision of Drebin – DT appears to be impacted differently in 6 groups while SVM in 4 groups. There is no impact observed for RF and KNN.

Table 9 further summarizes the accuracy of the experiments. Accuracy is usually considered to be one of the most important metrics for checking if a machine learning classifier can be adopted in practice, as the cost of errors can be huge, e.g., requiring huge efforts for practitioners to review the results manually. Interestingly, as highlighted by the Δ column, which calculates the difference between two experimental settings (e.g., E2-E1), for almost all of the cases, the differences are positive. The machine learning classifiers trained based on a dataset without duplicated samples are generally more accurate than such classifiers that are trained with datasets containing duplicated samples. A possible explanation for this effect could be that, compared to the latter case, the former setting contains more diverse malware samples, i.e., more malware characteristics, which

Table 9. Accuracy of the classification with different experimental settings.

| Dataset | Type | Setting | # Features | Δ | # Duplicated Vectors (Ratio) | Accuracy(%) SVM | Δ | DT | Δ | KNN | Δ | RF | Δ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Drebin | Dex | E1 (Without Duplication) | 34,799 | -3,500 | 581 (23.32%) | 94.99 | -0.2 | 92.27 | -1.56 | 91.1 | -1.51 | 92.88 | -1.64 |
| | | E2 (With Duplication) | 31,299 | | 1,341 (53.85%) | 94.79 | | 90.71 | | 89.59 | | 91.24 | |
| | Opcode seq | E3 (Without Duplication) | 33,398 | -3,555 | 255 (12.74%) | 93.64 | 0.02 | 89.91 | -1.83 | 88.62 | -1.29 | 91.42 | -0.61 |
| | | E4 (With Duplication) | 29,843 | | 864 (43.17%) | 93.66 | | 88.08 | | 87.33 | | 90.81 | |
| | API call | E5 (Without Duplication) | 30,289 | -3,347 | 206 (11.12%) | 93.63 | 0.04 | 87.96 | 0.24 | 88.02 | -0.12 | 90.29 | -0.73 |
| | | E6 (With Duplication) | 26,942 | | 756 (40.81%) | 93.67 | | 88.2 | | 87.9 | | 89.56 | |
| AMD | Dex | E1 (Without Duplication) | 135,015 | -23,666 | 2,745 (20.42%) | 98.55 | 0.03 | 92.36 | -0.49 | 96.13 | -1.49 | 92.48 | -0.24 |
| | | E2 (With Duplication) | 111,349 | | 5,838 (43.42%) | 98.58 | | 91.87 | | 94.64 | | 92.24 | |
| | Opcode seq | E3 (Without Duplication) | 107,270 | -16,522 | 909 (10.1%) | 98.22 | -0.06 | 96.42 | -0.92 | 95.61 | -0.94 | 94.98 | -0.43 |
| | | E4 (With Duplication) | 90,748 | | 3,058 (33.96%) | 98.16 | | 95.5 | | 94.67 | | 94.55 | |
| | API call | E5 (Without Duplication) | 101,418 | -15,763 | 585 (7.12%) | 98.28 | -0.01 | 95.94 | -0.63 | 95.55 | -0.52 | 95.7 | -1.48 |
| | | E6 (With Duplication) | 85,655 | | 2,569 (31.27%) | 98.27 | | 95.31 | | 95.03 | | 94.22 | |
| RmvDroid | Dex | E1 (Without Duplication) | 73,911 | -700 | 898 (14.43%) | 97.88 | 0 | 95.88 | 0.13 | 95.73 | -0.08 | 95.67 | -0.65 |
| | | E2 (With Duplication) | 73,211 | | 1,066 (31.79%) | 97.88 | | 96.01 | | 95.65 | | 95.02 | |
| | Opcode seq | E3 (Without Duplication) | 58,773 | -8,497 | 247 (5.76%) | 97.14 | 0.08 | 95.29 | -1.85 | 94.36 | -0.27 | 95.1 | -1.89 |
| | | E4 (With Duplication) | 50,276 | | 869 (20.28%) | 97.22 | | 93.44 | | 94.09 | | 93.21 | |
| | API call | E5 (Without Duplication) | 52,695 | -8,141 | 160 (4.19%) | 97.28 | 0.02 | 95.57 | -2.45 | 93.64 | -0.1 | 94.22 | -1.28 |
| | | E6 (With Duplication) | 44,554 | | 680 (17.82%) | 97.3 | | 93.12 | | 93.54 | | 92.94 | |

could make the classifier more powerful in locating new samples. Nevertheless, the performance difference is still within a small range, no matter which machine learning algorithm is used.

**RQ-3** Answer

Sample duplication can have diverse impacts on the performance of different machine learning models. However, no matter which machine learning algorithm is concerned, the impact seems to be marginal.

### 5.3 RQ4: Impact of Duplication Bias on Unsupervised Malware Clustering

In previous subsections, we have explored the impact of sample duplication on supervised learning approaches, w.r.t. three types of sample duplications that may have been overlooked by many state-of-the-art ML-based Android malware detection approaches. In this section we now explore the impact of sample duplication on unsupervised learning approaches, with a special focus on the duplication of malware families. The reason why unsupervised learning is selected is that it is one of the most common techniques used by researches to identify Android malware families [17, 18].

**Experimental Setup.** Fig. 11 illustrates the process we followed to prepare the training sets for setting up the experiments in answering this last research question. The first step for setting up the experiments is to perform malware family analysis to identify the family of a given malware. Fortunately, all the datasets have been released with family labels associated with their samples. In this work, we directly leverage those labels to conduct our experiments.

Given a dataset, once the family labels are identified for all its malware, we rank the families based on the number of malware they are assigned to. For the sake of simplicity, and to better present the experimental results, we choose the top-10 families to form our experiments. For the malware of the top-10 families, given a duplication type, we separate the malware samples into two sets: ND (all the samples are non-duplicated from each other) and D (all the samples are duplicated to that of ND), following the same strategy we leveraged for setting up the experiments in answering RQ2. Based on this distribution, we form 2 experiments: one without duplicated samples (i.e., $S1'$, the ND set is directly leveraged) while another does contain duplicated samples (i.e., $S2'$ contains samples from both ND and D and the size is equal to that of $S1'$).

Finally, for these three types of duplication, we set up six experiments, two for each duplication type and form a control group.

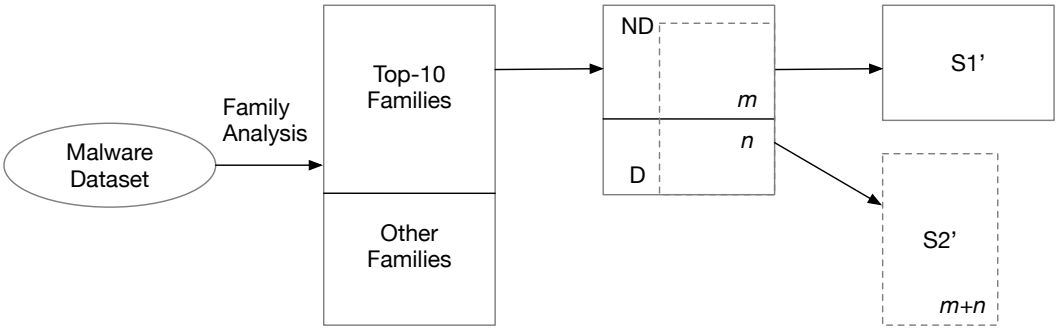


Fig. 11. Training Sets Preparation for ML-based malware family classification.

- $E1'/E2'$: Two datasets respectively contain or do not contain DEX Duplication, i.e., $E1'$ utilizes $S1'$ and $E2'$ utilizes $S2'$.
- $E3'/E4'$: Similar to $E1'/E2'$ except that opcode sequence duplication is used instead of DEX Duplication.
- $E5'/E6'$: Similar to $E1'/E2'$ as well except that in this time API call duplication is used instead of DEX Duplication.

Recall that our dataset for this experiment is formed by 10 families of apps. We hence set the final cluster numbers to 10, $k = 10$ for both K-means and Gaussian Mixture Model (GMM) [54] clustering algorithms[8], allowing for a better and clearer evaluation of the capability of the unsupervised learning models. In this setting, the learning model will group the input dataset into 10 clusters (e.g., clusters $1 \rightarrow 10$). Unfortunately, apart from grouping data samples into clusters, unsupervised learning approaches do not label the yielded clusters. To this end, after the clustering approach, we further leverage a straightforward approach to label the clusters. Specifically, given a cluster, we compare it to the inputted 10 family sets. We leverage the Jaccard similarity coefficient to calculate the distance between the given cluster and the original 10 malware family sets. Jaccard similarity coefficient is a simple yet well-known metric that has been frequently leveraged to calculate the similarity of sample sets, including the similarity of clusters categorized by unsupervised learning approaches [44, 53]. Given two clusters A, B, the Jaccard Index can be calculated as the ratio of the size of the intersection of A and B to the size of the union of A and B (cf. the formula below). The corresponding Jaccard index can be a value between 0 and 1, with 0 indicating no overlap (i.e., the two clusters are totally different) and 1 complete overlap (i.e., the two clusters are exactly the same) between the two clusters.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

**Results.** Table 10 summarizes the experimental results of applying unsupervised learning approaches to cluster Android malware families, concerning Dex duplication in the training samples (i.e., $E1'$ and $E2'$). The ten selected malware families are shown in the first row of the table and the clustering results are enumerated in the second column, simply named as cluster $1 \rightarrow 10$. The value in each cell shows the Jaccard index between the samples in a cluster and the samples in a given family. $Distance = 0$ indicates that there is no overlap between the cluster and the given family.

[8]Two of the most popular clustering algorithms. GMM can be regarded as an optimized version of the K-means model. In this work, we include two clustering algorithms to avoid potential biases.

Table 10. Jaccard distance between clustering results and the original family samples on Drebin.

| Setting | Cluster No. | FakeInstaller | DroidKungFu | Plankton | Opfake | GinMaster | BaseBridge | Iconosys | Kmin | FakeDoc | Geinimi | Label |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K-means E1' (Without Duplication) | Cluster 1 | **0.356** | 0 | 0 | 0.2218 | 0 | 0.0629 | 0.0031 | 0.0569 | 0 | 0 | FakeInstaller |
| | Cluster 2 | 0.0045 | 0.1726 | 0.1048 | 0.0155 | **0.2859** | 0.0866 | 0.0148 | 0 | 0.0065 | 0.1058 | GinMaster |
| | Cluster 3 | 0 | **0.1917** | 0 | 0 | 0.0115 | 0.0042 | 0 | 0 | 0 | 0 | DroidKungFu |
| | Cluster 4 | 0 | 0 | **0.7929** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Plankton |
| | Cluster 5 | 0 | 0 | 0 | 0 | 0 | 0 | **0.625** | 0 | 0 | 0 | Iconosys |
| | Cluster 6 | **0.2752** | 0 | 0 | 0.2208 | 0 | 0 | 0.1569 | 0 | 0 | 0 | FakeInstaller |
| | Cluster 7 | 0 | 0 | 0 | 0 | **0.2644** | 0 | 0 | 0 | 0 | 0 | GinMaster |
| | Cluster 8 | 0 | **0.4607** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | DroidKungFu |
| | Cluster 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0.881** | 0 | FakeDoc |
| | Cluster 10 | 0 | 0 | 0 | 0 | 0 | 0.205 | 0 | **0.5635** | 0 | 0 | Kmin |
| K-means E2' (With Duplication) | Cluster 1 | **0.5855** | 0 | 0 | 0.1261 | 0 | 0 | 0 | 0 | 0 | 0 | FakeInstaller |
| | Cluster 2 | 0.0327 | **0.297** | 0 | 0.0215 | 0.2336 | 0.0309 | 0.0036 | 0 | 0.0022 | 0 | DroidKungFu |
| | Cluster 3 | 0 | **0.5814** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | DroidKungFu |
| | Cluster 4 | 0 | 0 | **0.7127** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Plankton |
| | Cluster 5 | 0 | 0 | 0 | 0 | 0 | 0 | **0.625** | 0 | 0 | 0 | Iconosys |
| | Cluster 6 | 0 | 0.0133 | **0.2429** | 0 | 0.1939 | 0.0024 | 0 | 0 | 0 | 0.0027 | Plankton |
| | Cluster 7 | 0 | 0 | 0 | 0 | **0.2705** | 0 | 0 | 0 | 0 | 0 | GinMaster |
| | Cluster 8 | 0 | 0 | 0 | 0 | 0 | **0.6822** | 0 | 0 | 0 | 0 | BaseBridge |
| | Cluster 9 | 0 | 0 | 0 | 0 | 0 | 0.085 | 0 | 0 | 0.2929 | **0.55** | Geinimi |
| | Cluster 10 | 0.0859 | 0 | 0 | 0.2628 | 0 | 0.0079 | 0.1571 | **0.336** | 0 | 0 | Kmin |
| GMM E1' (Without Duplication) | Cluster 1 | 0.0064 | 0.0934 | 0.0622 | 0.0586 | **0.5248** | 0.0561 | 0.0091 | 0.0172 | 0.0091 | 0.0174 | GinMaster |
| | Cluster 2 | 0 | 0 | **0.6902** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Plankton |
| | Cluster 3 | **0.4676** | 0 | 0 | 0.1501 | 0 | 0.0022 | 0.0968 | 0.2104 | 0 | 0 | FakeInstaller |
| | Cluster 4 | 0 | 0 | 0 | 0.0035 | 0 | 0 | **0.4658** | 0 | 0 | 0.3646 | Iconosys |
| | Cluster 5 | 0 | **0.159** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | DroidKungFu |
| | Cluster 6 | 0 | **0.1405** | 0 | 0 | 0 | 0.0049 | 0 | 0 | 0 | 0 | DroidKungFu |
| | Cluster 7 | 0 | 0 | 0 | 0 | 0 | **0.5273** | 0 | 0 | 0.2791 | 0 | BaseBridge |
| | Cluster 8 | 0 | 0 | **0.1926** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Plankton |
| | Cluster 9 | 0 | **0.5164** | 0 | 0 | 0.0016 | 0 | 0 | 0 | 0 | 0 | DroidKungFu |
| | Cluster 10 | 0 | 0 | 0 | **0.1368** | 0 | 0 | 0 | 0 | 0 | 0 | Opfake |
| GMM E2' (With Duplication) | Cluster 1 | 0 | 0.0655 | 0.0021 | 0.0045 | **0.5714** | 0.0407 | 0.0147 | 0 | 0.0054 | 0.0149 | GinMaster |
| | Cluster 2 | 0 | 0 | **0.3579** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Plankton |
| | Cluster 3 | **0.4711** | 0 | 0 | 0.082 | 0 | 0.0021 | 0.0734 | 0 | 0.1153 | 0.2051 | FakeInstaller |
| | Cluster 4 | 0 | 0 | 0 | 0 | 0 | 0 | **0.6316** | 0 | 0 | 0 | Iconosys |
| | Cluster 5 | 0 | **0.3839** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | DroidKungFu |
| | Cluster 6 | 0 | **0.5174** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | DroidKungFu |
| | Cluster 7 | 0 | 0 | 0 | 0.0994 | 0 | **0.449** | 0 | 0.3761 | 0 | 0 | BaseBridge |
| | Cluster 8 | 0 | 0 | **0.6388** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Plankton |
| | Cluster 9 | 0 | 0 | 0 | 0 | **0.2705** | 0 | 0 | 0 | 0 | 0 | GinMaster |
| | Cluster 10 | 0.0044 | 0 | 0 | **0.3684** | 0 | 0 | 0.0743 | 0 | 0 | 0 | Opfake |

For each cluster, we calculate its Jaccard indexes to all malware families and label it based on the family that achieves the largest index value. Let us take the second row as an example, for cluster 1 in $E1'$, we calculate 10 Jaccard indexes respectively for the 10 considered malware families, among which we obtain five positive indexes. Since the largest index goes to FakeInstaller, i.e., samples in this cluster are closer to the FakeInstaller family than others, we label this cluster as FakeInstaller.

To better present the difference between the two settings – with or without duplicated samples – we visualize the results in Fig. 12. Each ellipse represents a malware family. If the family is no longer identified after clustering, it will be highlighted with dotted lines. The families are connected through directed edges. Each directed edge represents a mis-clustering from source family to the target family. The mis-clustered sample numbers are further displayed as the weight of the edge, and which is also reflected by the thickness of the edge. For example, in the graph of $E1'$ in Fig. 12, there is an edge from *Kmin* to FakeInstaller. The weight (14/85) indicates that there are 14 out of 85 Kmin malware being recognized as FakeInstaller malware. Fig. 13 further illustrates the visualization of misclassified malware families for all the other experimental settings (for the results returned by K-means only, the results returned by GMM are more or less the same and hence are not displayed to save space). From these visualized experimental results, Table 12 and Table 13 highlight the major differences returned by K-means and GMM, respectively.

Finally, we leverage MoJoFM [62], a Mojo distance based effectiveness metric, to measure the effectiveness of the selected two clustering models. The MoJoFM metric provides a more objective evaluation of the performance of clustering approaches, and can provide a single number as output that is simple to interpret and compare. Table 11 summarizes the experimental results. No matter which datasets or experimental settings are concerned, the differences between the MoJoFM scores achieved by K-means and GMM are not significant. However, no matter which clustering algorithms

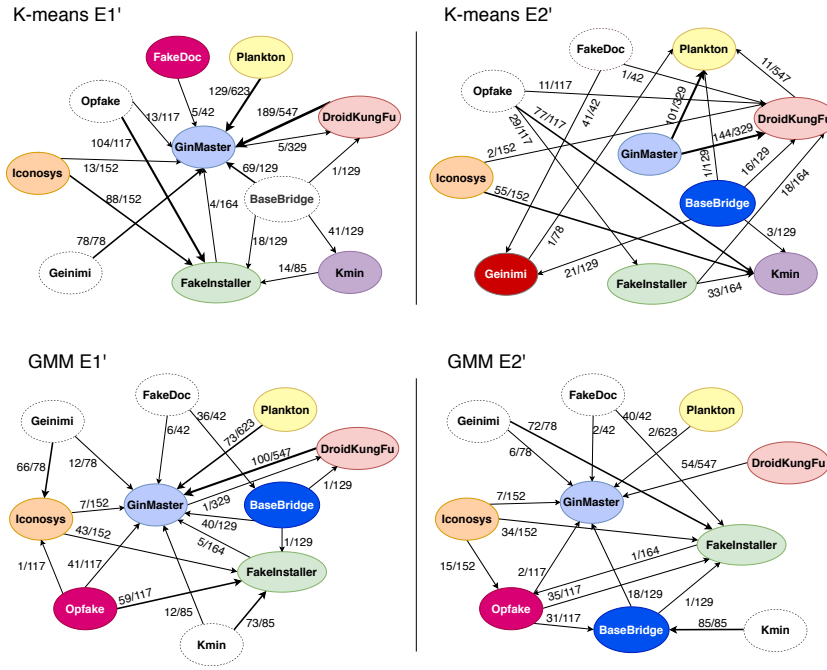


Fig. 12. Visualization of the misclassified malware families (for $E1'/E2'$ of the Drebin dataset). The coloured shapes represent the retained families. The shapes with dotted lines represent such families that are not identified by the clustering approach.

Table 11. The MoJoFM Distance between the clustering results and the original partition.

| Type | Setting | Drebin | | AMD | | RmvDroid | |
|---|---|---|---|---|---|---|---|
| | | K-means(%) | GMM(%) | K-means(%) | GMM(%) | K-means(%) | GMM(%) |
| Dex | E1' (Without Duplication) | 67.64 | 69.68 | 83.42 | 75.69 | 66.4 | 67.76 |
| | E2' (With Duplication) | 76.24 | 75.62 | 79.11 | 77.72 | 65.85 | 68.35 |
| Opcode seq | E3' (Without Duplication) | 70.07 | 76.35 | 81.26 | 81.75 | 57.72 | 61.81 |
| | E4' (With Duplication) | 73.76 | 78.34 | 75.02 | 74.55 | 58.38 | 58.83 |
| API call | E5' (Without Duplication) | 79.4 | 77.38 | 93.15 | 80.21 | 61.34 | 59.33 |
| | E6' (With Duplication) | 82.2 | 78.57 | 72.62 | 77.36 | 58.69 | 58.16 |

are concerned, there is a clear difference between a controlled pair of experimental settings (e.g., without or with duplicated samples).

**Finding-4.1:** Sample duplication can indeed impact the performance of unsupervised learning models. For example, as shown in Fig. 12, given the same unsupervised learning model, e.g., K-means, the same malware dataset is clustered into 7 and 8 families (i.e., the colored shapes) for $E1'$ and $E2'$ (without and with dex duplication), respectively.

**Finding-4.2:** The unidentified malware families are almost always different between the two experimental settings in a controlled pair. Indeed, as shown in the fifth column of Table 12 and Table 13, only one out of 9 pairs achieves the same result. This evidence suggests that, unlike that of supervised learning models, the impact of sample duplication on unsupervised learning models is quite significant.

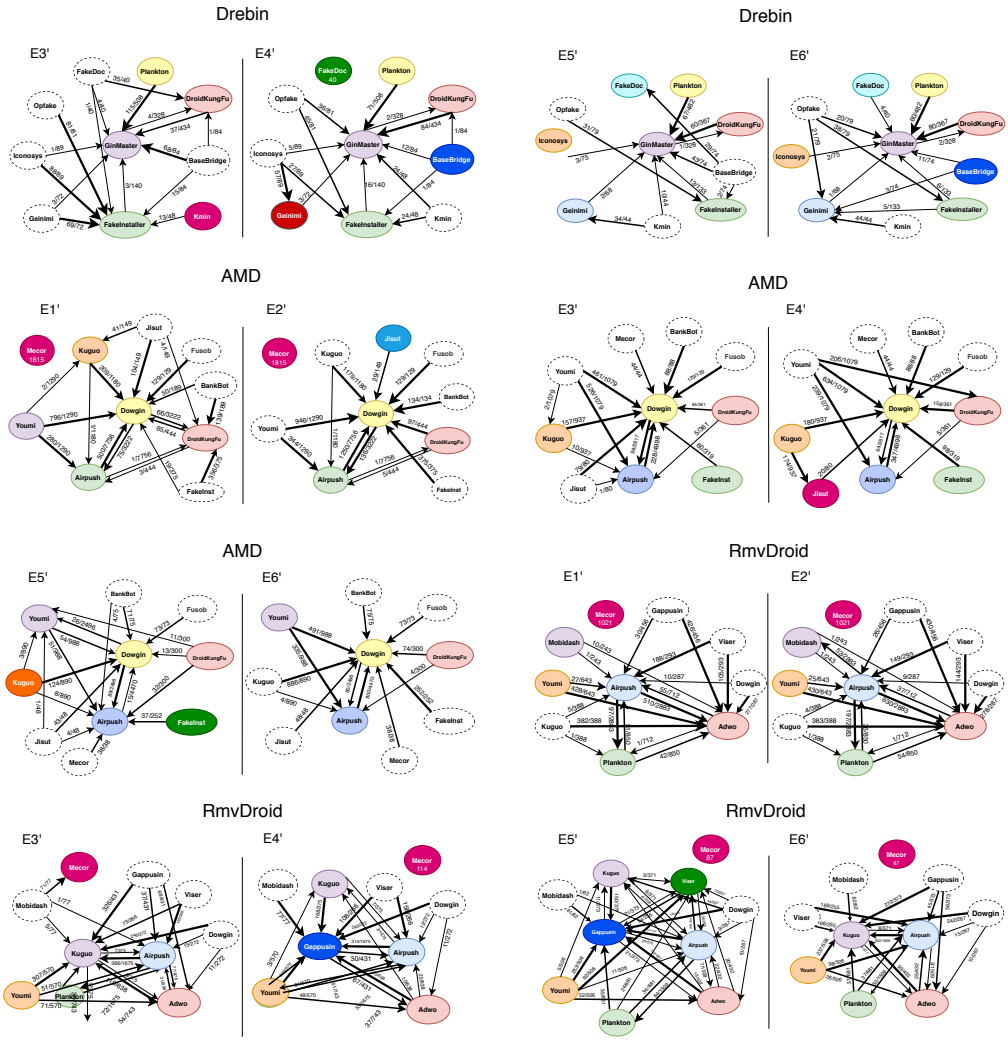


Fig. 13. Visualization of the misclassified malware families (for all the other experimental settings).

**Finding-4.3:** The impact of sample duplication in unsupervised learning based malware classifications is independent of clustering algorithms. Likely, no matter which clustering algorithms are selected, the clustering results will be impacted by experimental settings without or with duplicated samples.

Table 12. Summary of major misclassified results with K-means. Since the number of samples selected from the Drebin dataset is smaller than the other two datasets, the number of errors obtained for the Drebin dataset is also fewer than that of the others.

| Dataset | Type | Setting | #.Mistakes (Ratio) | Unidentified Families | Top 3 Error |
|---|---|---|---|---|---|
| Drebin | Dex | E1' (Without Duplication) | 771 (34.02%) | BaseBridge, Geinimi, Opfake | DroidKungfu-189/547->GinMaster, Plankton-129/623->GinMaster, Opfake-104/117->FakeInstaller |
| | | E2' (With Duplication) | 565 (24.93%) | FakeDoc, Opfake | GinMaster-144/329->DroidKungFu, GinMaster-101/329->Plankton, Opfake-77/117->Kmin |
| | Opcode seq | E3' (Without Duplication) | 538 (29.5%) | BaseBridge, FakeDoc, Geinimi, Iconosys, Opfake | Plankon-115/508->GinMaster, Iconosys-88/89->FakeInstaller, Opfake-81/81->FakeInstaller |
| | | E4' (With Duplication) | 408 (22.37%) | Iconosys, Kmin, Opfake | DroidKungFu-84/434->GinMaster, Plankton-71/508->GinMaster, Iconosys-57/89->Geinimi |
| | API call | E5' (Without Duplication) | 295 (17.46%) | BaseBridge, Kmin, Opfake | Plankton-67/482->GinMaster, DroidKungFu-60/367->GinMaster, BaseBridge-43/74->GinMaster |
| | | E6' (With Duplication) | 297 (17.57%) | Kmin, Opfake | DroidKungFu-80/367->GinMaster, Plankton-60/482->GinMaster, Kmin-44/44->Geinimi |
| AMD | Dex | E1' (Without Duplication) | 2,821 (17.05%) | BankBot, FakeInst, Fusob, Jisut | Youmi-796/1290->Dowgin, Airpush-500/7756->Dowgin, FakeInst-356/375->DroidKungFu |
| | | E2' (With Duplication) | 4,626 (27.95%) | BankBot, FakeInst, Fusob, Kuguo, Youmi | Airpush-1250/7756->Dowgin, Kuguo-1179/1180->Dowgin, Youmi-946/1290->Dowgin |
| | Opcode seq | E3' (Without Duplication) | 1,953 (18.33%) | BankBot, Fusob, Jisut, Mecor, Youmi | Youmi-526/1079->Airpush, Youmi-461/1079->Dowgin, Airpush-228/4998->Dowgin |
| | | E4' (With Duplication) | 2,386 (22.4%) | BankBot, Fusob, Mecor, Youmi | Youmi-634/1079->Dowgin, Airpush-347/4998->Dowgin, Youmi-239/1079->Airpush |
| | API call | E5' (Without Duplication) | 697 (7.24%) | BankBot, Fusob, Jisut, Mecor | Kuguo-124/890->Dowgin, Dowgin-89/2496->Airpush, Fusob-73/73->Dowgin |
| | | E6' (With Duplication) | 3,140 (32.61%) | BankBot, FakeInst, Fusob, Jisut, Kuguo, Mecor | Kuguo-886/890->Dowgin, Airpush-800/4470->Dowgin, Youmi-491/988->Dowgin |
| RmvDroid | Dex | E1' (Without Duplication) | 2,646 (34.05%) | Dowgin, Gappusin, Kuguo, Viser | Airpush-510/2883->Adwo, Youmi-428/643->Adwo, Gappusin-426/456->Adwo |
| | | E2' (With Duplication) | 2,889 (37.18%) | Dowgin, Gappusin, Kuguo, Viser | Airpush-630/2883->Adwo, Youmi-430/643->Adwo, Gappusin-430/456->Adwo |
| | Opcode seq | E3' (Without Duplication) | 2,180 (42.27%) | Dowgin, Gappusin, Mobidash, Viser | Airpush-366/1675->Kuguo, Gappusin-326/431->Kuguo, Youmi-307/570->Kuguo |
| | | E4' (With Duplication) | 1,924 (37.31%) | Dowgin, Mobidash, Viser | Youmi-329/570->Gappusin, Airpush-314/1675->Gappusin, Dowgin-249/272->Gappusin |
| | API call | E5' (Without Duplication) | 1,738 (38.29%) | Dowgin, Mobidash | Airpush-275/1509->Gappusin, Youmi-253/506->Gappusin, Dowgin-194/267->Gappusin |
| | | E6' (With Duplication) | 1,879 (41.4%) | Dowgin, Gappusin, Mobidash, Viser | Airpush-302/1509->Kuguo, Gappusin-272/373->Kuguo, Dowgin-242/267->Kuguo |

**RQ4** Answer

Sample duplication has an impact on the performance of unsupervised machine learning models. The impact can be observed in all of our experiments with either different malware datasets or different duplication types. Furthermore, unlike supervised learning for which insignificant impact is observed, the impact of sample duplication on unsupervised learning is quite significant, and such an impact is independent of the underline selected learning algorithms.

## 6 DISCUSSION

We now discuss the results of supervised learning with feature section (Subsection 6.1) and the experiment setting with realistic malware/goodware distribution in the test set (Subsection 6.2). We

Table 13. Summary of major misclassified results with GMM.

| Dataset | Type | Setting | #.Mistakes (Ratio) | Unidentified Families | Top 3 Error |
|---|---|---|---|---|---|
| Drebin | Dex | E1' (Without Duplication) | 577 (25.46%) | Kmin, FakeDoc, Genimi | DroidKungfu-100/547->GinMaster, Kmin-73/85->FakeInstaller, Plankton-73/623->GinMaster |
| | | E2' (With Duplication) | 405 (17.87%) | Kmin, FakeDoc, Genimi | DroidKungFu-54/547->GinMaster, Kmin-85/85->BaseBridge, Genimi-72/78->FakeInstaller |
| | Opcode seq | E3' (Without Duplication) | 386 (21.16%) | Opfake, BaseBridge, Iconosys, Kmin | GinMaster-64/328->DroidKungFu, Iconosys-65/89->GinMaster, Opfake-51/81->Genimi |
| | | E4' (With Duplication) | 517 (28.34%) | Opfake, Iconosys, Kmin, FakeDoc, Genimi | DroidKungFu-89/434->GinMaster, Geinimi-72/72->GinMaster, Plankton-52/508->GinMaster |
| | API call | E5' (Without Duplication) | 411 (24.32%) | Opfake, BaseBridge, Kmin, Genimi | DroidKungFu-109/367->GinMaster, BaseBridge-67/74->GinMaster, Genimi-64/68->FakeInstaller |
| | | E6' (With Duplication) | 389 (23.02%) | Iconosys, Kmin, Genimi | GinMaster-91/328->DroidKungFu, Iconosys-74/75->Opfake, Genimi-66/68->Opfake |
| AMD | Dex | E1' (Without Duplication) | 4,258 (25.73%) | FakeInst, Fusob, BankBot, Jisut, DroidKungFu | Dowgin-1265/3222->Kuguo, Youmi-632/1290->Kuguo, Airpush-536/7756->Kuguo |
| | | E2' (With Duplication) | 3,277 (19.8%) | Youmi, Fusob, BankBot, Jisut, DroidKungFu | Youmi-734/1290->Airpush, Dowgin-705/3219->Kuguo, Youmi-529/1288->Dowgin |
| | Opcode seq | E3' (Without Duplication) | 3,465 (32.53%) | Mecor, Kuguo, BankBot, Jisut, DroidKungFu | Airpush-828/4998->Youmi, Kuguo-600/937->Dowgin, Dowgin-467/2617->Youmi |
| | | E4' (With Duplication) | 2,345 (22.01%) | Mecor, Fusob, Kuguo, BankBot, Jisut | Kuguo-691/937->Youmi, Airpush-460/4998->Youmi, Youmi-256/1079->Dowgin |
| | API call | E5' (Without Duplication) | 1,969 (20.45%) | Mecor, Fusob, Kuguo, BankBot, Jisut | Kuguo-830/890->Dowgin, Airpush-381/4470->Dowgin, Youmi-201/988->Dowgin |
| | | E6' (With Duplication) | 2,487 (25.83%) | Mecor, Youmi, BankBot, Jisut | Youmi-524/988->Dowgin, Airpush-438/4470->Dowgin, Kuguo-255/890->Dowgin |
| RmvDroid | Dex | E1' (Without Duplication) | 2,612 (33.62%) | Dowgin, Gappusin, Viser | Airpush-614/2883->Kuguo, Gappusin-345/456->Kuguo, Youmi-320/643->Kuguo, |
| | | E2' (With Duplication) | 2,423 (31.18%) | Dowgin, Gappusin, Kuguo | Kuguo-365/388->Airpush, Gappusin-309/456->Airpush, Airpush-277/2879->Viser |
| | Opcode seq | E3' (Without Duplication) | 2,115 (41.01%) | Dowgin, Gappusin, Mobidash, Viser | Gappusin-302/431->Kuguo, Airpush-265/1675->Kuguo, Dowgin-247/272->Kuguo |
| | | E4' (With Duplication) | 2,169 (42.06%) | Dowgin, Mobidash, Mecor, Gappusin | Gappusin-285/431->Kuguo, Dowgin-241/272->Kuguo, Airpush-240/1675->Kuguo |
| | API call | E5' (Without Duplication) | 1,870 (41.2%) | Dowgin, Mobidash | Kuguo-328/371->Airpush, Youmi-241/506->Gappusin, Dowgin-231/267->Airpush |
| | | E6' (With Duplication) | 2,173 (47.87%) | Dowgin, Gappusin, Adwo | Airpush-269/1507->Viser, Gappusin-263/373->Kuguo, Dowgin-240/267->Kuguo |

then stress the importance of considering sample duplication for machine learning (Subsection 6.3), and summarize the effect of parameter turning for ML-based malware detectors (Subsection 6.4) and the potential threats to validity of our study (Subsection 6.5).

## 6.1 Supervised Learning with Feature Selection

Recall that we have directly leveraged the features proposed by Arp et al. [8] to evaluate the impact of sample duplication in machine learning based Android malware detection. The set of features eventually considered in this work is hence comprehensive, which may subsequently overfit the learning algorithm. Towards verifying this hypothesis, we replicate one of our previous experiments by integrating feature selection into the working process. Specifically, for the experiment conducted in Section 5.1.2, after the full feature set is extracted, we introduce a feature selection step into our approach aiming at retaining only such features that have importance weights higher than a given

Table 14. Experimental results (over the Drebin dataset) with feature selection applied. The features are selected only if their importance weights are higher than the average weights calculated based on the full feature set.

| Type | Setting | $\|ND\|$ | Training Set | Test Set | # Original Features | # Selected Features | Malware | | | Goodware | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Precision(%) | Recall(%) | F1 score(%) | Precision(%) | Recall(%) | F1 score(%) |
| Dex | E1 (Without Duplication) | 3,559 | 2,491*2 | 1,068*2 | 34,799 | 10,485 | 92.6 | 97.28 | 94.88 | 97.14 | 92.22 | 94.62 |
| | E2 (With Duplication) | 3,559 | 2,491*2 | 1,068*2 | 31,299 | 8,970 | 93.62 | 95.07 | 94.34 | 94.99 | 93.52 | 94.26 |
| Opcode seq | E3 (Without Duplication) | 2,859 | 2,001*2 | 858*2 | 33,398 | 10,024 | 90.5 | 96.73 | 93.51 | 96.49 | 89.85 | 93.05 |
| | E4 (With Duplication) | 2,859 | 2,001*2 | 858*2 | 29,843 | 8,789 | 92.23 | 94.78 | 93.48 | 94.63 | 92.01 | 93.3 |
| API call | E5 (Without Duplication) | 2,646 | 1,852*2 | 794*2 | 30,289 | 9,250 | 92.42 | 95.33 | 93.85 | 95.18 | 92.18 | 93.66 |
| | E6 (With Duplication) | 2,646 | 1,852*2 | 794*2 | 26,942 | 8,011 | 92.99 | 94.36 | 93.67 | 94.28 | 92.89 | 93.58 |

threshold. In this experiment, we set the threshold to be the average weights calculated based on the full feature set. Table 14 illustrates the new experimental results. The seventh column presents the numbers of selected features, which are significantly smaller (over 50%) than that of the original features (as shown in the sixth column). Nonetheless, by comparing with the experimental results shown in Table 8 (that obtained without involving feature selection), the experimental results are not significantly impacted by involving feature selection to the process. This evidence suggests that the selection of a large number of features has a limited impact on the experimental results of this work.

### 6.2 Realistic Malware/Goodware Distribution in Test Set

For all the experiments conducted in the evaluation section, we have followed many of the existing works by fulfilling the test datasets with balanced apps (i.e., containing the same number of malware and goodware). Unfortunately, this setting does not reflect the actual distribution of malware/goodware in the real world. Subsequently, the corresponding experimental results may not be able to represent the actual performance achievable in practice. We hence design additional experiments to check if such more realistic settings will impact our experimental findings. To the best of our knowledge, there is no ground truth about the actual distribution of malware/goodware, and it is non-trivial to obtain that in practice. Pendlebury et al. [52] have attempted to estimate such a ratio based on samples collected from the public AndroZoo dataset, which contains over 8 million apps at the time of their study. Eventually, they conclude that a reasonable estimation of malware to goodware distribution could be 1:9.

In this work, we take this distribution ratio to fulfil the additional experiments, i.e., by preparing new test/training datasets. For the sake of simplicity, since only the Drebin dataset has been provided with benign samples, we replicate the experiment (as presented in Section 5.1.2) on the Drebin dataset only. The machine learning models are trained with the same algorithm and with the same dataset when a balanced training dataset is concerned or with the newly prepared unbalanced dataset. Table 15 summarizes the new experimental results. Interestingly, for the experiment of *Unbalanced Training Set, Unbalanced Test Set*, the experimental results are comparable to that achieved by *Balanced Training Set, Balanced Test Set*. When a balanced training set is concerned, i.e., *Balanced Training Set, Unbalanced Test Set*, while retaining very high recall of detecting malware as such, the precision has significantly decreased compared to the experimental results achieved by an unbalanced training set or a balanced test set. Nevertheless, similar to the findings we summarized previously, based on these new experimental results, differences can still be observed between such experiments trained with or without duplicated samples. This result once again suggests that sample duplication should be carefully considered (and avoided) when performing machine learning based Android malware detection.

Table 15. Experimental results of SVM-based malware classification obtained based on realistic malware/-goodware distribution (i.e., 1:9).

| Type | Setting | Training Set | Test Set | # Features | # Duplicated Vectors (Ratio) | Malware | | | Goodware | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Precision(%) | Recall(%) | F1 score(%) | Precision(%) | Recall(%) | F1 score(%) |
| Balanced Training Set, Unbalanced Test Set | | | | | | | | | | | |
| Dex | E1 (Without Duplication) | 2,491*2 | 1,068+9,612 | 34,799 | 581 (23.32%) | 58.36 | 97.19 | 72.93 | 99.7 | 93.08 | 96.28 |
| | E2 (With Duplication) | 2,491*2 | 1,068+9,612 | 31,299 | 1,341 (53.85%) | 63.66 | 95.06 | 76.25 | 99.48 | 94.58 | 96.97 |
| Opcode seq | E3 (Without Duplication) | 2,001*2 | 858+7,722 | 33,398 | 255 (12.74%) | 60.75 | 96.62 | 74.59 | 99.54 | 92.16 | 95.71 |
| | E4 (With Duplication) | 2,001*2 | 858+7,722 | 29,843 | 864 (43.17%) | 65.2 | 94.74 | 77.24 | 99.3 | 93.65 | 96.39 |
| API call | E5 (Without Duplication) | 1,852*2 | 794+7,146 | 30,289 | 206 (11.12%) | 59.53 | 95.33 | 73.29 | 99.36 | 91.77 | 95.41 |
| | E6 (With Duplication) | 1,852*2 | 794+7,146 | 26,942 | 756 (40.81%) | 62.68 | 94.31 | 75.31 | 99.23 | 92.87 | 95.94 |
| Unbalanced Training Set, Unbalanced Test Set | | | | | | | | | | | |
| Dex | E1 (Without Duplication) | 2,491+22,419 | 1,068+9,612 | 34,799 | 581 (23.32%) | 90.82 | 91.75 | 91.28 | 99.18 | 99.07 | 99.13 |
| | E2 (With Duplication) | 2,491+22,419 | 1,068+9,612 | 31,299 | 1,341 (53.85%) | 92.16 | 88.5 | 90.29 | 98.86 | 99.25 | 99.05 |
| Opcode seq | E3 (Without Duplication) | 2,001+18,009 | 858+7,722 | 33,398 | 255 (12.74%) | 88.93 | 90.9 | 89.9 | 98.85 | 98.58 | 98.72 |
| | E4 (With Duplication) | 2,001+18,009 | 858+7,722 | 29,843 | 864 (43.17%) | 90.87 | 86.76 | 88.76 | 98.34 | 98.91 | 98.63 |
| API call | E5 (Without Duplication) | 1,852+16,668 | 794+7,146 | 30,289 | 206 (11.12%) | 90.9 | 89.41 | 90.15 | 98.66 | 98.86 | 98.76 |
| | E6 (With Duplication) | 1,852+16,668 | 794+7,146 | 26,942 | 756 (40.81%) | 92.34 | 86.07 | 89.09 | 98.25 | 99.09 | 98.67 |

### 6.3 The importance of sample duplication for machine learning.

In this work, we experimentally show that sample duplication indeed impacts the performance of machine learning-based Android malware detection approaches, w.r.t. both supervised and unsupervised learning models. This result aligns with the results reported by Miltiadis Allamanis in investigating the adverse effects of code duplication in machine learning models of code [4]. In this work, we would like to emphasize that the impact of duplication on Android malware detection is quite marginal for supervised ML approaches. Unfortunately, the rationale behind this marginal impact is unclear at the moment. In our future work, we plan to fill this gap by conducting advanced explainable machine learning techniques.

Nonetheless, we argue that sample duplication could introduce biases depending on the ML-based classification approaches that may be used. We hence advocate that practitioners and researchers should pay more attention to sample duplication in their ML-based classifications. Ideally, sample duplications could be taken as a machine learning parameter, which needs to be explicitly communicated when reporting the performance of given machine learning approaches. Indeed, just like any other parameters of ML algorithms, such as $k$ for the K-means algorithm, sample duplication rate is essential for supporting the reproducibility of the ML approaches. To help practitioners and researchers better communicate the sample duplications in their datasets for that of Android-oriented approaches, we further present to the community a prototype tool for characterizing duplicated samples in an Android app dataset. This tool further provides options for users to exclude duplicated samples from their datasets. We have made our prototype tool available online at https://github.com/carol233/duplication.

### 6.4 The effect of parameter turning for ML-based malware detectors

As empirically revealed by Allix et al. [5] in their large-scale empirical assessment of machine learning based malware detectors, no matter in which settings – 10-fold cross-validation or training on one set and test another set – RandomForest always achieves the best precision compared with other machine learning algorithms (including C4.5, RIPPER, SVM [5]). This empirical finding, surprisingly, is different from the one that we observed in this work. We hence go one step deeper to check the possible reasons behind this difference. We followed the "Drebin" approach to set up our experiments, for which an additional "grid-search" step is adopted by searching for suitable parameter values for our learning algorithms. This is in contrast to the approach of Allix et al., who simply used the default options. Therefore, in this work, we re-launch all the experiments with the "grid-search" feature disabled. In this circumstance, RandomForest indeed jumps up to be the best learning algorithm for predicting Android malware. This contradictory result suggests that it is

vital to tune ML algorithm parameter values when performing machine learning based malware classification. The algorithm that works best out of the box in default mode may not be the most suitable one if parameter turning is concerned [16, 24, 58]. This implication is further backed up by the fact that SVM rather than RaondomForest is adopted by the "Drebin" approach, although the RandomForest algorithm is frequently reported as one of the best algorithms in the literature.

### 6.5 Threats to Validity

The main threat to *construct validity* of our study concerns the exhaustiveness of classification algorithms we selected and the experiments we set up in this work. Although we have selected four well-known algorithms and both in-the-lab and in-the-wild experimental settings, which have also been frequently leveraged by other researchers to achieve similar purposes, they may not be entirely suitable for predicting Android malware [59]. Nonetheless, the four algorithms yield more or less similar results suggest that our findings are not specific to a particular learning algorithm. Another threat to *construct validity* lies in the process of preparing training/test datasets [58]. In this work, to ensure a balance between the size of training and test datasets, we choose a threshold of 30% to form the test set. This threshold may not be representative of this study. Ideally, to be fully conclusive, we would need to experiment with more thresholds. However, this is not the main focus of this work, we leave it as future work. Furthermore, when preparing the training and testing datasets for evaluating the impact of sample duplication for supervised learning approaches, as shown in Fig. 6, there might be chances that some samples in the testing set have their duplicated counterparts set in the training set. This setting may lead to slightly higher classification performance as the malware detector could learn some malware information in advance. A more realistic setting would be to limit the testing samples to not include duplicated versions of the apps in the training set. An ideal approach could be to take app release time into consideration when preparing the training/testing set, e.g., testing apps are all released after the testing set, which is an ideal situation since the malware detector cannot learn from future samples, as suggested by Li et al [40]. Nevertheless, this is also not the main focus of this paper, we leave it as future work.

Yet another threat to *construct validity* concerns the feature extraction process of this work. Recall that, in this work, we directly leverage the feature set of the "Drebin" approach to train our machine learning models. However, the authors do not make their feature extraction scripts publicly available. To this end, we had to resort to the re-implementation of Annamalai Narayanan[9] to extract features from Android apps. Our re-implementation may not be identical to that of the original authors. Nonetheless, our re-implementation has been successfully adopted by both the authors themselves and many of our fellow researchers working in this community [49, 64]. Furthermore, the features extracted by the "Drebin" approach are mainly based on syntactic rules (e.g., the appearance of certain strings), which may not be able to characterize the semantic features of Android apps. Subsequently, the machine learning results might be impacted. In our future work, we plan to alleviate this impact by leveraging semantic features such as the ones extracted based on Android apps' graph representations [21] and advanced deep learning algorithms such as the ones driven by neural networks.

The key threat to *internal validity* concerns possible errors in the implementation of our experimental tools and scripts used to run the experiments and gather experimental results. To reduce this threat, we have carefully reviewed the code and scripts of our toolchain to ensure that the implemented functions meet our expectations. We have further manually checked a random selection of experimental results to verify their accuracy.

[9]https://github.com/MLDroid/drebin

One threat to the *external validity* of our study concerns the representativeness of the malware datasets that we selected. Although we have included four common malware datasets from the literature, our results may still not be generalisable to other malware datasets. Nonetheless, the fact that our experimental findings are similar among the selected datasets shows that the external validity of our work is likely to be reasonable. Also, to avoid potential biases, we restrict the test dataset to contain unduplicated samples only when conducting the supervised learning experiments, which unfortunately may not reflect the real-world situation as it is likely to have duplicated samples in a real-world dataset. Nevertheless, since this decision will not impact the capability of the classifier (which only relies on the training dataset) and the duplication rate in practice is not significant, such a decision should only bring limited threats to our experiments and hence could be neglected. In this work, the performance of our family clustering experiments is directly related to the authenticity of malware labels, which unfortunately may not be reliable, as often discussed by other researchers [25, 30, 57]. To mitigate this threat, we have directly leveraged the malware labels provided by the malware datasets, which have already been leveraged by various prior research projects.

## 7 RELATED WORK

Machine learning-based Android malware classification has been a hot topic in the software engineering and security community. Below we summarize some representative prior work.

**Android malware detection.** Machine learning has been extensively leveraged by practitioners and researchers to detect Android malware [50, 66]. One of the most common algorithms leveraged by researchers for achieving this purpose is RandomForest, which has been reported by researchers as one of the best algorithms for conducting binary classification. As an example, Alam et al. [3] have empirically demonstrated that RandomForest is optimal by comparing its accuracy with BayesNet, Logistic Regression, DT, etc. Later on, Allix et al. [5] have also empirically confirmed this. In their experiment, they experimentally show that RF achieves the best performance compared with C4.5, RIPPER, and SVM. A similar result has also been backed up by Li et al. [38] as well. In this work, although different datasets and feature sets are concerned, we achieve more or less similar results, i.e., RandomForest is among the best algorithm for precisely discriminating malware from goodware.

As discussed in the previous section, with "grid-search" enabled to optimise parameters, SVM in many cases can achieve an even better performance than that of RandomForest. Hence, SVM has also been a very common machine learning algorithm for training to predict Android malware. For example, Naser et al. [51] built a malware detector based on the features statically extracted from Android APKs. One of the most famous works that leverage SVM to predict Android malware is the one presented by Arp et al. [8]. They proposed the Drebin approach, for which they extract machine learning features from Android APKs (or DEX files) into eight feature sets. In our work, aiming at exploring the effect of sample duplication on machine learning based malware detectors, we leveraged the same feature sets and included SVM as one of our four evaluated machine learning algorithms. In many of our experimental settings, SVM indeed performs the best compared to that of other machine learning algorithms.

Most of the aforementioned approaches extract features statically from Android DEX files, which contain the core app code of the apps. In our work we have thus empirically explored the impact of DEX duplication on machine learning approaches. Apart from the DEX file, we have also included two extra duplication types involving app opcode and API calls. These two types have been frequently leveraged by other researchers to form feature sets for learning the malicious behaviors of Android apps. Indeed, as an example, Jerome et al. [32] have proposed an ML-based malware detection approach based on opcode sequences in 2014. Similar to their work, Canfora

et al. [15] and McLaughlin et al. [47] also respectively present machine learning based malware detection approaches based on features statically extracted from the raw Dalvik bytecode (i.e., opcode sequences). Since similar opcode sequences can be extracted from different apps, i.e., opcode sequence duplication, the performance of the approaches mentioned above might be impacted.

Similar to opcode sequences, Android APIs have also been recurrently leveraged as features for machine learning based Android malware analysis. For example, in 2013, Aafer et al. [2] have performed a thorough analysis that leverages critical API calls as features to evaluate the difference among selected classification algorithms. In their experiments, they employed four algorithms, including DT, C4.5, KNN, and linear SVM. Their experimental results reveal that KNN is the best algorithm for predicting malware when API calls are considered as features. This finding is quite different from ours as we experimentally show that RF and SVM are among the best algorithms. We note that our experiments are done on different datasets and use different feature sets, although API calls are considered by both approaches. Similarly, in 2016, Wu et al. [63] leveraged the use of dataflow-related API-level features to improve the performance of a KNN detector. We observe that approaches of leveraging API calls as features may be impacted by API call duplication if the authors do not carefully sanitize their training dataset.

In addition to traditional machine learning models, researchers have also started to leverage deep learning models to detect Android malware. In 2014, Yuan et al. [68] built a deep learning model with more than 200 features extracted from both static and dynamic analysis and stated that deep learning techniques are especially applicable for Android malware detection. Likewise, in 2018, Karbab et al. [33] proposed an Android malware detection and family identification framework, MalDozer, which also leverages deep learning techniques to predict Android malware. In this work, we only focus on investigating the effect of sample duplication on traditional machine learning models. We nonetheless believe deep learning models are also relevant to the sample duplication concerns that we highlighted in this work. We plan to explore this direction in our future work.

**Android malware family classification.** In addition to machine learning-based malware detection, practitioners and researchers have also spent a significant amount of effort to identify the family of Android malware [27]. For example, Garcia et al. [27] proposed a novel approach for detecting malware families. By leveraging features extracted from specific Android API usages, reflective calls, and native binaries, they designed and implemented a prototype tool RevealDroid to achieve this purpose.

Most state-of-the-art approaches leverage unsupervised learning to identify Android malware families. The rationale behind this is that similar malware (belonging to the same family) will be grouped into the same cluster. As an example, Bayer et al. [12] have identified and grouped malware exhibiting similar behavior with a scalable clustering method. Similarly, in 2013, Hu et al. [29] designed and implemented a framework, namely MutantX-S, to cluster samples into families based on code instruction sequences efficiently. They have also proven that MutantX-S is highly accurate in detecting previously unknown malware. In 2015, Aresu et al. [7] created Android malware clusters by analyzing specific statistical information related to the HTTP traffic.

The above papers used clustering methods to aggregate malware with similar malicious behavior, which is of great significance for obtaining the family classification labels of malware. Unfortunately, none of these approaches has taken into account the sample duplication problem in their experimental setting, and thereby their performance might not be reliable.

**Bias in machine learning.** Apart from applying machine learning techniques to characterize Android malware, researchers have also started to investigate the potential biases that appear in the working processes of machine learning-based techniques. Pendlebury et al. [52] recently presented a study discussing the potential biases in two dimensions: space (referred to as spatial bias) and time (referred to as temporal bias). Spatial bias is caused by the unrealistic setting of the ratio of

benign to malware samples in training and test data. Temporal bias refers to the integration of future knowledge about test data into the training stage. Similarly, Li et al. [40] have experimentally shown that time inconsistency introduces significant biases to machine learning based malware detection approaches.

In 2018, Li et al. [36] presented a study demonstrating that more features used by a machine learning approach do not necessarily mean better performance. In a recent work reported by Irolla et al. [31], 49.35% of the samples in the Drebin dataset have at least one more sample containing the same sequence of opcode. This result is in line with the findings of this work. Indeed it actually motivated us to investigate the potential impact of such duplication on the performance of machine learning approaches.

To the best of our knowledge, our work is the first to investigate the impact of sample duplication on machine learning-based Android malware detection approaches. However, studies on the adverse effects of code duplication in machine learning models have also been carried out. Allamanis et al. [4] presented a technical report describing the impact of multiple file-level (near-)clones appearing in large corpora of code. They discussed the biases introduced mathematically and empirically proved that code duplication can lead to overestimating the performance when evaluating machine learning models. Different from their work, our work in this paper targeting Android malware at the bytecode level.

## 8 CONCLUSION

In this paper, we empirically investigated the impact of sample duplication on machine learning-based Android malware detection approaches. We started by recognizing common sample duplication types in well known and used Android malware datasets. We then took into account these sample duplication types to train distinctive machine learning models to classify Android malware. We conducted our experiments on three common malware datasets. Our experimental results show that sample duplication does indeed impact the performance of machine learning-based malware detection approaches. An in-depth exploration further revealed that this finding applied to not only in-the-lab experiments (i.e., 10-fold cross-validation) but also in-the-wild analyses (i.e., trained on one dataset and then tested on another). This finding also applies to experiments that were conducted using different machine learning algorithms, including both supervised and unsupervised learning approaches.

## 9 ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers who have provided insightful and constructive comments to help in improving this paper. This work was supported by the Australian Research Council (ARC) under a Laureate Fellowship project FL190100035, a Discovery Early Career Researcher Award (DECRA) project DE200100016, and a Discovery project DP200100020, by the National Natural Science Foundation of China (No. 61702045), by the Fonds National de la Recherche (FNR), Luxembourg, under project CHARACTERIZE C17/IS/11693861, by the SPARTA project, which has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 830892.

## REFERENCES

[1] 2019. Sequential minimal optimization. https://en.wikipedia.org/wiki/Sequential_minimal_optimization

[2] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*. Springer, 86–103.

[3] Mohammed S Alam and Son T Vuong. 2013. Random forest classification for detecting android malware. In *2013 IEEE international conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing*. IEEE, 663–669.

[4] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 143–153.

[5] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2016. Empirical assessment of machine learning-based malware detectors for Android. *Empirical Software Engineering* 21, 1 (01 Feb 2016), 183–211. https://doi.org/10.1007/s10664-014-9352-6

[6] 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 '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[7] Marco Aresu, Davide Ariu, Mansour Ahmadi, Davide Maiorca, and Giorgio Giacinto. 2015. Clustering android malware families by http traffic. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 128–135.

[8] 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 the Network and Distributed System Security Symposium (NDSS), San Diego, CA*, Vol. 14. 23–26.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[10] Zarni Aung and Win Zaw. 2013. Permission-based android malware detection. *International Journal of Scientific & Technology Research* 2, 3 (2013), 228–234.

[11] 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 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 426–436.

[12] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, behavior-based malware clustering.. In *NDSS*, Vol. 9. Citeseer, 8–11.

[13] Daniel Bilar. 2007. Opcodes as predictor for malware. *International journal of electronic security and digital forensics* 1, 2, 156–168.

[14] Evgeny Burnaev and Dmitry Smolyakov. 2016. One-class SVM with privileged information and its application to malware detection. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. IEEE, 273–280.

[15] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. 2015. Mobile malware detection using op-code frequency histograms. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, Vol. 4. IEEE, 27–38.

[16] Joymallya Chakraborty, Tianpei Xia, Fahmid M Fahid, and Tim Menzies. 2019. Software Engineering for Fairness: A Case Study with Hyperparameter Optimization. *arXiv preprint arXiv:1905.05786* (2019).

[17] Tanmoy Chakraborty, Fabio Pierazzi, and VS Subrahmanian. 2017. EC2: ensemble clustering and classification for predicting android malware families. *IEEE Transactions on Dependable and Secure Computing* (2017).

[18] Luke Deshotels, Vivek Notani, and Arun Lakhotia. 2014. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014*. ACM, 3.

[19] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. FraudDroid: Automated Ad Fraud Detection for Android Apps. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*.

[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] Ming Fan, Xiapu Luo, Jun Liu, Meng Wang, Chunyin Nong, Qinghua Zheng, and Ting Liu. 2019. Graph embedding based familial analysis of android malware using unsupervised learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 771–782.

[22] Ming Fan, Wenying Wei, Xiaofei Xie, Yang Liu, Xiaohong Guan, and Ting Liu. 2020. Can We Trust Your Explanations? Sanity Checks for Interpreters in Android Malware Analysis. *arXiv preprint arXiv:2008.05895* (2020).

[23] Ivan Firdausi, Alva Erwin, Anto Satriyo Nugroho, et al. 2010. Analysis of machine learning techniques used in behavior-based malware detection. In *2010 second international conference on advances in computing, control, and telecommunication technologies*. IEEE, 201–203.

[24] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *Information and Software Technology* 76 (2016), 135–146.

[25] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. 2019. Should You Consider Adware as Malware in Your Study?. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2019)*.

[26] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. 2020. Borrowing Your Enemy's Arrows: the Case of Code Reuse in Android via Direct Inter-app Code Invocation. In *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*.

[27] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 3 (2018), 11.

[28] Benjamin Holland, Tom Deering, Suresh Kothari, Jon Mathews, and Nikhil Ranade. 2015. Security toolbox for detecting novel and sophisticated android malware. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 733–736.

[29] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. 2013. Mutantx-s: Scalable malware clustering based on static features. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 187–198.

[30] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. 2017. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for Android malware. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 425–435.

[31] Paul Irolla and Alexandre Dey. 2018. The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques* 14, 3 (2018), 245–249.

[32] Quentin Jerome, Kevin Allix, Radu State, and Thomas Engel. 2014. Using opcode-sequences to detect malicious Android applications. In *2014 IEEE International Conference on Communications (ICC)*. IEEE, 914–919.

[33] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. 2018. MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation* 24 (2018), S48–S59.

[34] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* (2018).

[35] Chenglin Li, Keith Mills, Di Niu, Rui Zhu, Hongwen Zhang, and Husam Kinawi. 2019. Android malware detection based on factorization machine. *IEEE Access* 7 (2019), 184008–184019.

[36] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-an, and Heng Ye. 2018. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3216–3225.

[37] Li Li. 2017. Mining androzoo: A retrospect. In *The Doctoral Symposium of 33rd International Conference on Software Maintenance and Evolution (ICSME-DS 2017)*.

[38] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. 2015. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *The 2015 IEEE International Conference on Software Quality, Reliability & Security (QRS)*.

[39] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*.

[40] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2018. MoonlightBox: Mining Android API Histories for Uncovering Release-time Inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*.

[41] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2019. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering (TSE)* (2019).

[42] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* (2017).

[43] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. 2017. AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. *arXiv preprint arXiv:1709.05281* (2017).

[44] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. 2017. On locating malicious code in piggybacked android apps. *Journal of Computer Science and Technology* 32, 6 (2017), 1108–1124.

[45] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé F Bissyandé, and Jacques Klein. 2020. MadDroid: Characterising and Detecting Devious Ad Content for Android Apps. In *The Web Conference 2020 (WWW 2020)*.

[46] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Network and Distributed Systems Security Symposiym (NDSS)*. San Diego, CA.

[47] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. 2017. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 301–308.

[48] Stuart Millar, Niall McLaughlin, Jesus Martinez del Rincon, Paul Miller, and Ziming Zhao. 2020. DANdroid: A Multi-View Discriminative Adversarial Network for Obfuscated Android Malware Detection. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. 353–364.
[49] Annamalai Narayanan, Guozhu Meng, Liu Yang, Jinliang Liu, and Lihui Chen. 2016. Contextual Weisfeiler-Lehman graph kernel for malware detection. In *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 4701–4708.
[50] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps.. In *NDSS*.
[51] Naser Peiravian and Xingquan Zhu. 2013. Machine learning for android malware detection using permission and api calls. In *2013 IEEE 25th international conference on tools with artificial intelligence*. IEEE, 300–305.
[52] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 729–746. https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury
[53] Roberto Perdisci and ManChon U. 2012. VAMO: towards a fully automated malware clustering validity analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 329–338.
[54] Douglas A Reynolds. 2009. Gaussian Mixture Models. *Encyclopedia of biometrics* 741 (2009).
[55] S Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics* 21, 3 (1991), 660–674.
[56] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. 2013. Puma: Permission usage to detect malware in android. In *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*. Springer, 289–298.
[57] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 230–253.
[58] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering* (2018).
[59] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.
[60] Haoyu Wang, Junjun Si, Hao Li, and Yao Guo. 2019. RmvDroid: towards a reliable Android malware dataset with app metadata. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 404–408.
[61] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*. Springer, Bonn, Germany, 252–276.
[62] Zhihua Wen and Vassilios Tzerpos. 2004. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 194–203.
[63] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. 2016. Effective detection of android malware based on the usage of data flow APIs and machine learning. *Information and Software Technology* 75 (2016), 17–25.
[64] Zhiwu Xu, Kerong Ren, Shengchao Qin, and Florin Craciun. 2018. CDGDroid: Android Malware Detection Based on Deep Learning Using CFG and DFG. In *International Conference on Formal Engineering Methods*. Springer, 177–193.
[65] Wei Yang, Mukul R Prasad, and Tao Xie. 2018. Enmobile: Entity-based characterization and analysis of mobile malware. In *Proceedings of the 40th International Conference on Software Engineering*. 384–394.
[66] Xinli Yang, David Lo, Li Li, Xin Xia, Tegawendé F Bissyandé, and Jacques Klein. 2017. Characterizing malicious Android apps by mining topic-specific data flow signatures. *Information and Software Technology* (2017).
[67] Jian Yu, Miin-Shen Yang, and E Stanley Lee. 2011. Sample-weighted clustering methods. *Computers & mathematics with applications* 62, 5 (2011), 2200–2208.
[68] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 371–372.
[69] XU Zhiwu, Kerong Ren, and Fu Song. 2019. Android Malware Family Classification and Characterization Using CFG and DFG. In *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 49–56.
[70] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109.