Experimental Study with Real-world Data for Android App Security Analysis using Machine Learning

Sankardas Roy[1], Jordan DeLoach[2], Yuping Li[3], Nic Herndon[2], Doina Caragea[2], Xinming Ou[3],
Venkatesh Prasad Ranganath[2], Hongmin Li[2], and Nicolais Guevara[2]

[1]Computer Science Department, Bowling Green State University

[2]Department of Computing and Information Sciences, Kansas State University

[3]Department of Computer Science and Engineering, University of South Florida

ABSTRACT

Although Machine Learning (ML) based approaches have shown promise for Android malware detection, a set of critical challenges remain unaddressed. Some of those challenges arise in relation to proper evaluation of the detection approach while others are related to the design decisions of the same. In this paper, we systematically study the impact of these challenges as a set of research questions (*i.e.*, hypotheses). We design an experimentation framework where we can reliably vary several parameters while evaluating ML-based Android malware detection approaches. The results from the experiments are then used to answer the research questions. Meanwhile, we also demonstrate the impact of some challenges on some existing ML-based approaches. The large (market-scale) dataset (benign and malicious apps) we use in the above experiments represents the real-world Android app security analysis scale. We envision this study to encourage the practice of employing a better evaluation strategy and better designs of future ML-based approaches for Android malware detection.

1. INTRODUCTION

Android devices and apps are becoming increasingly popular. According to a recent Gartner report [1], Android devices held an 80.7% share in the 2014 sales of smartphones, totaling nearly one billion units. The most popular app store is Google Play holding more than one million apps and having a user population of more than one billion. The huge size of the Android ecosystem also attracts many attackers. As many security companies have recently reported [13, 17], the Android ecosystem is infected with malware. These malware apps do various nefarious activities, such as stealing user credentials and private information, sending text messages to premium numbers, *etc.* This is a major problem with high stakes warranting immediate attention.

One direction for detection of Android malware apps is to tap the power of Machine Learning (ML) algorithms. Several efforts explored this direction and proposed solutions [5, 6, 9, 21] in the recent past. A typical ML-based approach for Android malware app detection (*ML-approach* henceforth) employs a classifier (*e.g.*, an off-the-shelf ML classifier, such as *k*-NN) which is trained by a *training set* consisting of known benign apps and known malware apps. To evaluate the classification performance, the number of correctly and incorrectly classified apps is measured on a *test set* whose labels are unknown to the classifier at the time of evaluation. Although the results of the ML-inspired approaches look promising, many critical research questions still remain unanswered. There exists substantial room for clarification and improvement.

In this paper we investigate challenges that are faced in applying ML for Android malware app detection. We have studied the existing ML-approaches, and found that they all are affected by one or more challenges as discussed below.

Challenges in ensuring proper evaluation: These challenges arise in selecting the evaluation metrics as well as in collecting and preparing the data (*e.g.*, correctly labelling the apps in training/test set). We see that most of the current ML-approaches are impacted by related factors: (a) the evaluation strategy does not follow a common standard; (b) the ground truth on which these approaches are evaluated lack reliability.

Challenges related to algorithm design: These challenges arise in the design space of the ML-approaches. For instance, one challenge is to construct an informative *feature set* for the classifier. For example, in the Drebin work [5] the feature set contains hundreds of thousands of items, and many items (such as the names of the app components) are arbitrary strings at app developer's choice. This raises a question on whether all items in this large feature set are really helping the classifier or if a subset can be sufficient (or even better).

The previously proposed ML-approaches focused more on a specific setting defined by factors such as specific evaluation metrics, ground truth quality, composition of the training/test data, the feature set, and others. The reported performance results are then measured in the particular setting. However, since the setting varies widely across different approaches, it is difficult (if not impossible) to fairly compare

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA
© 2015 ACM. ISBN 978-1-4503-3682-6/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818038

the results. For many of the recently proposed solutions, we are not aware of the impact of the above factors on the classifier performance.

To systematically study the impact of the aforementioned factors, we formulate a set of research questions (RQs) which we answer in this paper. We refer to these factors as *impact factors* (IFs) from now on. To facilitate our study, we designed an experimental framework that has the capability of evaluating a given ML-approach. Furthermore, we designed a candidate ML-approach named *c-approach* so that we can reliably vary the parameters (corresponding to IFs) while evaluating it on our experimental framework. The results from these experiments in turn answer the RQs.

Through our experiments we show that the IFs play a substantial role in the performance evaluation results. We argue that we have to take into account the IFs if we want to compare different ML-approaches in an unbiased way. Further, to the best of our knowledge, ours is the first ML-based work that studies a large scale (Android market-scale) data, which includes nearly 1 million Google Play apps, and about 250K adware/malware apps. The largest benign dataset considered by a previous ML-approach had about 120K apps [5] while the largest malicious dataset had 24K apps [6]. We strongly believe that our large data set is substantially better representative of the real world.

Our overarching contribution lies in our attempt to make the research community aware of a set of critical challenges in designing a ML-approach for Android malware detection. In particular, our main contributions are as follows.

- We formulated a set of research questions (RQs) hypothesizing the role of the impact factors (IFs), such as (a) the metrics used, (b) the characteristics of the base data set, (c) algorithm sophistication, *etc.* on an ML-approach's evaluation results.

- We designed an experimental framework and extensively studied our *c-approach* in different settings. The results of our experiments give answer to the RQs.

- Our work is the first to study a market-scale dataset in the context of ML-based Android malware detection.

2. CHALLENGES IN APPLYING ML FOR ANDROID MALWARE DETECTION

We formulate a set of research questions (RQs) hypothesizing the influence of the impact factors (IFs) that stem from the challenges in applying ML in Android malware detection. Below we categorize the challenges and discuss them in more details. For each such category, we introduce the list of corresponding RQs, which we experimentally study later in this paper.

2.1 Challenges in ensuring proper evaluation

To ensure that the ML-approach is evaluated properly is not straightforward. The related challenges fall under two subcategories as follows.

Challenges in deciding the evaluation metrics. The evaluation metrics for an ML-approach are not yet standardized and different ML-approaches rely on different metrics[1]. For instance, DroidSIFT [21] and MUDFLOW [6] report the performance results in terms of true positive rate (TPR) and false-positive rate (FPR). Other existing works, such as MAST [9] and Drebin [5] present the Receiver Operating Characteristic (ROC) plot, which is a generalized representation of TPR and FPR while the separating threshold is varied. Further, the ML-community has reported [10] that if the dataset is highly imbalanced, the PRC (precision-recall curve) is a better metric for measuring classifier performance than the traditional ROC curve. Given that the Android malware domain is highly imbalanced, *i.e.*, the ratio of malware to benign apps in the real-world is highly skewed (1:100 or up), the above facts raise substantial doubt on whether current works are using the best metric.

- RQ1: Is ROC the best metric to evaluate ML-based malware detection approaches?

Challenges due to characteristics of the input data. These challenges are related to data preparation, *e.g.*, labelling the apps, composing the training/testing set, and so on. We see that these challenges are applicable to all the current ML-approaches. For instance, the age of input data may pose one challenge. Dated apps vs. recent apps could lead to very different evaluation results in some cases. Deciding the data composition is another challenge, *e.g.*, selecting the ratio between positive class (malware apps) size and negative class (benign apps) size in the test data, which may lead to different performance results of the classifier. To ensure realistic evaluation, we should conform to the real-life ratio of malware and good apps in the app store, but unfortunately this is not practiced in many existing works. Furthermore, the ground truth is noisy in reality while manually labelling million plus apps is not feasible. So, we have to depend on security companies' report on those apps (if available), which effectively lead to imperfect ground truth. We see that the ground truth on which the current ML-approaches depend on is not fully reliable, which has a negative impact in two ways: (i) if training data has noise (mislabeled apps), the classifier mislearns things, which will negatively influence the classification performance. (ii) if test data has noise, we evaluate on wrong ground truth, and then, the reported performance results can be misleading. In addition, presence of adware apps (which show unwanted advertisements to the user) in the dataset leads to further challenges. As adware has similarities to both benign and malware apps, it is often challenging to label an adware, *e.g.*, including adware in the malware set, or in the benign set, or dropping adware from the dataset altogether. The existing works differ on this choice, which further complicates attempting to compare their performance.

- RQ2: Does the age of malware in training/test set mislead performance?

- RQ3: Does classifier performance degrade as we approach real-world ratio of malware and benign apps?

- RQ4: Does quality of ground truth affect the performance?

- RQ5: Does presence of adware in the dataset affect the performance?

[1]The definition of the commonly used metrics are available in Section 3.3

2.2 Challenges in the algorithm design

These challenges are related to the design of the ML-approach itself. One challenge is to construct an informative (*i.e.*, discriminative across the classes) feature set for the classifier. Some of the existing approaches are overwhelmed by this challenge. As an example, the Drebin approach [5] uses a very large feature set. One may want to know whether the classifier really needs this large feature set or only a subset of these items could be sufficient. We note that the size of Drebin's feature set is correlated with the size of its' dataset—it has nearly 500K features while applied on the authors' dataset [5], but when we emulated Drebin feature's extraction on our larger dataset we achieved more than 1 million features. Do we really need these many features? How to identify and select strong, discriminative features is a challenge.

- RQ6: Are more features always better?

2.3 Challenges in data collection

We have discussed above the challenges due to characteristics of the dataset. Collecting a large dataset of apps itself poses a formidable challenge. Attempting to collect modern apps is an even more challenging task. Although Google Play provides the whole set of 'free' apps (over 1.4 million), there is no 'download API' available. So, we need to rely upon app store crawlers like *PlayDrone* [18] that periodically scan the Google Play app store and collect entire snapshots of the store. The very most recent apps, however, are not always available in the PlayDrone archive. Moreover, collecting a large set of adware and malware apps, is also challenging – we have to rely on several sources. VirusShare and anti-virus companies provide large datasets of potentially malicious apps. These sets, however, are often noisy and impure, sometimes containing benign apps, Win32 binaries, and even blank apps. We believe that the large amount of data, even if somewhat noisy, provides further credibility to our results. To reduce the computation complexity of the ML-approach is a further challenge. It is not straightforward how to design a scalable ML-approach. When considering the millions of apps in the Play store, and the thousands of new apps added every day, scalability is of paramount importance. As an example of the degree of this challenge, we take note of MUDFLOW [6] authors' comment that sometimes their system took more than 24 hours to analyze one single Android app.

3. EXPERIMENTAL FRAMEWORK

We designed an experimental framework to study the challenges faced by the existing ML-based Android malware detection approaches. Our framework utilizes the ML-suite Weka [12] which provides a set of off-the-shelf classifier tools (*e.g.*, SVM, *k*-NN, *etc.*). To examine the research questions we pose, we also designed a candidate ML-approach named *c-approach*. The *c-approach* uses the *k*-NN classifier. The experimental framework puts into consideration a number of factors such as ground truth preparation, feature construction, and evaluation, which enables it to examine different ML-approaches. While studying a particular ML-approach, we choose the corresponding data preparation strategy, feature construction algorithm, and evaluation strategy. Below we discuss how we set up our experimental framework to study an ML-approach, which we explain mainly with the example of *c-approach*.

3.1 Ground truth preparation

We collected Android apps from app stores like Google Play store, as well as malware archives like VirusShare, and other third party companies like Arbor Networks. Specifically, we made use of the app archive built by an existing Android app crawling tool named PlayDrone [18] that collected approximately 1.4 million Android apps from Google Play in October 2014. We also gathered about 35K potentially malicious apps from VirusShare and 24K potentially malicious apps from Arbor Networks.

In order to prepare reliable ground truth data, we utilized VirusTotal [3], which is a free online service that can be used to check the maliciousness of files and URLs. In particular, when an app is queried against VirusTotal, it uses different (currently as many as 54) anti-virus (AV) products and scan engines to check for potential maliciousness. If the app has been scanned before, a user can query by the hash value (*e.g.*, md5 or sha1) of the app and could instantly get an aggregated scan report from the anti-virus companies. Among all the apps mentioned above, we found that VirusTotal already has reports for about 1 million apps. Considering the apps with no malicious label as benign and those with at least one malicious label as malicious, we ended up with about 855K benign apps and 247K malicious apps.

```
Data: VirusTotal reports for all the collected apps
Result: Benign app set $B$, malware app set $M$, adware
        app set $A$
foreach $report_i \in VirusTotal\ reports$ do
    Let $n$ = number of malicious flags in $report_i$;
    if $n == 0$ then
        Add $app_i$ in set $B$;
    end
    else if $n > 0$ then
        Let $a$ = number of adware labels in $report_i$;
        Let $m$ = number of malware labels in $report_i$;
        if $a > m$ then
            Add $app_i$ in set $A$;
        end
        else
            Add $app_i$ in set $M$;
        end
    end
end
```

Algorithm 1: Separation algorithm for benign app, malware, and adware

Within those malicious apps, we note that the apps can be further divided into adware and malware. We divide them into malware and adware to study their features separately as well as to prioritize the correct identification of malware apps above all else. The detailed scheme for separating benign apps, malware, and adware is described in Algorithm 1. Particularly, for the purpose of distinguishing malware and adware, we prepare a keyworld list for each category. For instance, adware keywords contain common words indicating 'ad' (*e.g.*, 'adware, 'AdDisplay', 'MultiAds', *etc.*) and common ad companies (*e.g.*, 'Airpush', 'RevMob', *etc.*). Malware keyword list includes words indicating maliciousness (*e.g.*, 'Trojan', 'Malware', 'Exploit', *etc.*) and the malware family names like 'DroidKungFu', 'FakeInst', *etc.* After processing all VirusTotal reports with the keywords list, Algorithm 1 provides us with about 141K adware, 106K malware, and 855K benign apps, and all of our subsequent experiments will select a subset of apps from this app universe.

We also note that more than half of our malicious apps are actually collected from the PlayDrone snapshot of the Google Play store. This constitutes about 20% of the apps from Google Play, which is significantly higher than 0.1% as reported in Google's 2014 Android security report [11]. The reasons for this difference could be: (1) Some of these problematic apps have been detected and removed from the Play store since the latest PlayDrone snapshot was taken; (2) VirusTotal reports contain false positives, especially for those apps that are detected as malicious by only a few AVs.

In order to study the impact of the quality of ground truth (*i.e.*, rate of false positives in the VirusTotal reports) on the ML-approach's performance results, we build different quality levels of ground truth.[2] In particular, we split malicious apps (including malware and adware) into different levels of certainty based on the number of malicious labels in the VirusTotal reports. The apps with higher number of malicious labels are considered as higher credibility apps, since the presence of maliciousness is verified by several different AVs. Particularly, we consider apps with greater than or equal to 10 malicious labels to be high quality malicious apps, and apps with only 1 malicious label as low quality ones. With this logic, we achieve 79K high quality malicious apps and 85K low quality malicious apps, and the rest of malicious apps are considered as neither high quality nor low quality apps. Note that for most of the RQs (*i.e.*, all but RQ4) we do not enforce any constraints regarding malware quality so as to represent a more real-world dataset.

3.2 Feature construction

To experiment with an ML-approach in our framework, we need to realize the feature extraction procedure as defined in that particular approach. In addition to implementing the *c-approach*, we also emulate the feature extraction of Drebin's [5] and DroidSIFT's [21]. We particularly chose these two ML-approaches because they provide two canonical examples: (i) Drebin is unique in terms of using huge number (in order of hundreds of thousands) of features; (ii) DroidSIFT is unique in using a feature vector that is based on graph distance related to API dependance graphs. We note that to answer most of the RQs (*i.e.*, all but RQ6) we utilize the *c-approach* while to answer RQ6, regarding quality of features, we make use of the emulated features of Drebin's and DroidSIFT's.

3.2.1 *For implementing* c-approach

Android is a privilege-separated operating system, in which each application operates in a process sandbox. It is required for each application to declare the permissions they require, and the Android system then prompts the user for consent at the time the application is installed. It has been shown [22] that a malware app often requests a pattern of permissions that are distinct from legitimate apps. This distinction can be used as indicators for potential malicious functionality. Particularly, we extract Android permission requests from the *Manifest file* and *intent action* strings from static analysis of disassembled code, and presence of '.so' or '.apk' files within the main application archive. After carefully studying the related Android documents and other ML-approaches (*e.g.*, MAST [9]), we have selected a set of 151 permissions, 112 *intent action* strings, and presence of '.so' or '.zip' in our feature set. Given an app, we extract these 265 features through a lightweight static analysis.

Since API level information conveys substantial semantics about app's behavior, researchers [5, 21] also proposed API level features (*e.g.*, method name, class name, *etc.*) to detect Android malware. We also use APIs as features while we select the set of critical APIs by studying 100K benign apps and 15K malicious apps. Specifically, we extract all the (uni-gram) method names from the disassembled app bytecode and calculate the overall usage ratio of each method in benign apps versus the malicious ones. We then select 101 methods that are most commonly associated with malicious apps and 100 methods most likely to be associated with benign apps. We note that an app typically contains a few third-party libraries and many apps share the same popular third-party libraries (web APIs, ad libraries, *etc.*). Thus, to capture the individual app-behavior, during the process of extracting uni-gram methods, we exclude those APIs that only appear within known third-party libraries.

According to our app analysis experiences, we observe that malware apps are often obfuscated. We conjecture that malware authors are more inclined (compared to benign apps) to obfuscate their apps since it helps disguise their malicious code. It is challenging to determine whether an app is obfuscated or not, and furthermore, if it is obfuscating proprietary information or malicious behavior. Nevertheless, we found certain information which can serve as a strong indicator of obfuscation. For example, class names like 'a.a.*' is a simple obfuscation technique that is used by default within many tools such as Proguard [14]. Additionally, certain Android obfuscators encrypt the original application file and decrypt it dynamically and load it via 'DexClassLoader' when executed. By studying obfuscating techniques, we further extend our feature set by including 5 features that are related to obfuscation.

In summary, we extract 471 features from the above 5 categories (permissions, *intent actions*, discriminative APIs, obfuscation signatures, native code signatures).

3.2.2 *For emulating Drebin features*

Drebin [5] is unique in terms of using huge number (in order of hundreds of thousands) of features while providing very impressive classification performance. Recall that we investigate in RQ6 whether all the features in the feature set are "informative." So, to answer RQ6 we are interested to study Drebin's features. Drebin authors do provide the feature vectors of their own dataset for evaluation by other researchers. However, the feature extraction program is not available while Drebin features are correlated with dataset. We thus develop our own tool to extract the same features based on the description in the Drebin paper, so that we can extract the Drebin-style feature vectors from the same base dataset we have used for our other experiments. Drebin's feature set depends on the input app set because many of Drebin's features are raw strings which appear in the manifest file or in the disassembled code of the app. In particular, Drebin extracts four types of features (hardware components, requested permissions, app components, and filtered

[2]We utilize these different ground truth datasets for the experiment in RQ4 as discussed in Section 4.2.

intents) from the manifest file, and another four types of features (restricted API calls, used permissions, suspicious API calls, URLs) from the disassembled code.

We could readily emulate the feature extraction for all types of the manifest features and one type of code features (URLs). We wrote Regular Expressions (RegEx) to match the style of these features and then feed those RegEx's into our scalable feature extractor. In addition, we emulated the extraction procedure of the remaining types of the 'code' features after we received further information (*i.e.*, entire list of 'suspicious API calls', and the 'API-permission mapping') from the Drebin authors via private communication.

We parallelized feature extractions across a High Performance Computing Cluster (HPCC) to extract features on our own dataset. Extracting Drebin-style features was a substantially more computationally complex process than our *c-approach* due to the sheer number of features extracted, per-app. We note that due to a larger base dataset, we also extracted a larger feature universe. We generated over one million features from only around 180K apps, and thus did not attempt to extract all features for our whole dataset.

3.2.3 For emulating DroidSIFT features

In addition to Drebin, we were inspired by the design of DroidSIFT [21] features, which are based on API dependency graph distances. Hence, DroidSIFT features are expected to be rich in semantics. So, to answer RQ6, we also chose to study the DroidSIFT features. As the DroidSIFT tool is not publicly available, we took help from (and worked with) the DroidSIFT authors to extract the feature vectors for our app set. In using this particular feature construction approach, we use a different reference database (from which the graph distances are computed) of graphs than the original DroidSIFT work, and our application of ML for malware detection is also different from DroidSIFT's use of the feature vectors to classify malware into families. Thus our study in no way reflects the quality of the DroidSIFT's original results. Our study is only to investigate whether all items in this unique type of features have good 'discriminative' power.

In particular, we created a dataset of 3,000 apps[3], subsampled from our main dataset. Among these 3K apps, one third are malware, one third adware, and the remaining are benign. Further, among them, 2,100 (*i.e.*, 700 apps from each class) were used for training, and 900 (*i.e.*, 300 apps from each class) were used for testing. As per our instruction, the DroidSIFT authors built a reference graph database using the above training set and returned us a feature vector for each app in our training set and test set. Each feature in the feature vector generated by DroidSIFT represents a graph distance of the candidate app from the reference graph database. From the 3,000 apps, DroidSIFT was able to generate the feature vectors only for 1,524 apps (1,128 from the training set and 396 from the test set). The remaining apps failed as they either were generating excessively large graphs (containing more than 100 nodes), taking too long to compute the graph edit distance, being developed for unsupported versions of Android, or could not be resolved to extract DEX executables.

[3]It only had 3000 apps since we seeked help from the DroidSIFT authors to generate these features for us and they were bound by computing resources to generate features for more than a few thousand apps.

3.3 Evaluation strategy

For all of our experiments, we employ cross-validation, which is a standard technique for assessing how the results of a statistical analysis will generalize to an independent data set. Specifically, for a k-fold cross validation, we randomly partition the original sample set into k equal sized subsamples while one of them is retained as the testing set and the remaining k - 1 subsamples are used as training data. The cross validation process is then repeated k times, with each of the k subsamples used exactly once as the test set, and the final k results are then averaged to produce an estimation. In particular, we use 5-fold cross-validation in our experiments. Since every sample in the datset is used once during evaluation, cross-validation can be used to avoid biased data points. However, we observe that cross validation technique was used by only a few of the recent ML-approaches [6, 20].

	actual positive (malware app)	actual negative (benign app)
predicted positive	true positive (TP)	false positive (FP)
predicted negative	false negative (FN)	true negative (TN)

Table 1: Confusion matrix

True positive rate = $\frac{TP}{TP+FN}$
False positive rate = $\frac{FP}{FP+TN}$
Precision = $\frac{TP}{TP+FP}$
Recall = $\frac{TP}{TP+FN}$

Table 2: Definition of metrics

In this research, we explore the applicability of the following evaluation metrics for the ML results. The metrics are defined using the standard confusion matrix (as in Table 1) as illustrated in Table 2. (i) true positive rate (TPR): the proportion of actual positives which are correctly identified as such; (ii) false positive rate (FPR): the proportion of actual negatives which are falsely identified as positive; (iii) precision: the fraction of returned positive results that are really positive; (iv) recall: same as TPR; (v) receiver operating characteristic curve (ROC) and area under the curve (auROC): ROC is a graphical plot that shows how TPR varies with FPR when the discrimination threshold is varied. The auROC represents measure of the area under the ROC curve; the closer the value of auROC getting to 1, the better the performance of the classifier. (vi) precision recall curve (PRC) and area under the curve (auPRC): PRC of (positive class) is a graphical plot that shows how precision varies with recall, when the discrimination threshold is varied; The auPRC measures the area under PRC curve. Similar to auROC, the closer the value of auPRC getting to 1, the better the performance of the classifier.

TPR (same as recall), FPR, and precision are computed at a certain discriminating threshold. One can obtain ROC and PRC curves by moving the threshold, thus the auROC and auPRC are more comprehensive metrics for a classifier's performance. Furthermore, many security researchers and practitioners observed [16] that an intrusion detection system is not practical unless we can limit the FPR within a bound. So, they advocate for studying 'ROC curve bounded by a FPR limit (say x)' instead of the regular ROC curve. As defined by prior researchers [15] auROC$_x$ represents 'normalized area under the ROC curve bounded by the $[0, x]$ interval on the FPR axis'. The normalization is done with respect to the maximum possible area, *i.e.*, $x \times 1$ (as the max TPR is 1). As noted by [15], auROC$_x$ enables the security practitioner to assess the detection rate because of a nice property: auROC$_x = b$ implies a TPR of b with a FPR $\leq x$.

4. EXPERIMENTATION

We have introduced (in Section 2) the research questions (RQs) which are to study the impact of the challenges that are typically faced by an ML-approach for Android malware detection. Recall that each RQ corresponds to an impact factor (IF), such as evaluation metric, data-composition, *etc.* To answer each RQ, we use our *c-approach* for two main reasons. (i) Most of the existing ML-approaches' implementations are not publically available; and (ii) Our in-house approach allows us to reliably vary the setting (*i.e.*, the IF) on our experimental framework to measure the impact of the corresponding factor. Furthermore, for some RQs we study some aspect of a few existing ML-approaches.

The implementation of our three part experimental framework begins with ground truth preparation. We use the techniques discussed in Section 3.1 to collect our base dataset from a variety of sources and partition it into malware, adware, and benign subsets. On a 48-core Ubuntu server, our multi-thread python implementation of Algorithm 1 can process about 1.5 million VirusTotal scan reports within 40 minutes. Then, to extract the features in the way described in Section 3.2, we utilize a High Performance Computing Cluster (HPCC) to individually examine and extract features from each app in the dataset. While extracting a feature vector from an app is relatively cheap and fast (only a few seconds), to do so for about one million apps is time-consuming if done sequentially. The HPCC enables us to scale to hundreds of cores and enables feature extractions on millions of apps in just a few hours. Our experimental framework is scalable and runs well on the HPCC, proving that our data-driven approach could handle the daily throughput of a major app store like Google Play. Then, we input the feature vectors extracted on the HPCC into Weka and choose the classifier (*e.g.*, k-NN). Note that Weka is a commonly used machine learning software suite boasting tens of algorithms and filters for both supervised and unsupervised learning. Finally, we run the experiments on Weka after selecting the corresponding evaluation metric(s). The training and evaluation of the classifier is substantially faster than feature extraction, and can be run in a matter of hours when parallelized on a common desktop machine.

In each experiment, we vary certain parameters while keeping the others constant. Unless otherwise stated, these are the defaults: (a) metrics: TPR, FPR, and auPRC of the malicious class, (b) malware/benign apps ratio in test data is 1:50, (c) data set size: 8.5K malware and 425K benign apps, (d) ML-approach: *c-approach*, *i.e.*, classifier being k-NN and feature set having 471 features, as described in Section 3.2. Note that we do not filter out adware from the malware set in any experiment except for in RQ4, *i.e.*, we allow adware being mixed with malware in all experiments except RQ4. Now we present the details of the experiments for each RQ. We classify the RQs in three categories as discussed below.

4.1 Impact of evaluation strategy

RQ1: Is ROC the best metric to evaluate ML-based malware detection approaches?

The most commonly used metric in existing literature in the domain of Android malware detection is currently the ROC curve [5, 9]. In this RQ we attempt to answer whether there could be a better metric for comparing different ML approaches especially for the domain of Android malware detection, where the ratio of malware to benign apps in the real-world may range from 1:100 and up [2].

If there is a large skew in input class dataset, the ROC curve can present an overly optimistic view of a classifier's performance [10]. On the other hand, PRC can expose differences between algorithms that are not apparent in ROC space. Since the number of negative samples (benign apps) usually significantly exceeds positive samples (malicious apps) in a legitimate app store (Google Play), the number of false positives is expected to be greatly smaller than true negatives. Thus, according to the metric definition of FPR as specified in Table 2, a large change in the number of false positives can only lead to a small change in FPR which is used in ROC analysis. However, since *precision* compares false positives against true positives as defined in Table 2, PRC will be able to capture the effect of the large number of negative examples on the algorithm's performance. Therefore, the performances of two algorithms may appear similar in ROC space, while in PRC space one may reflect a clear advantage compared to another one. In addition, PRC can reveal the effect of the base-rate fallacy [7] in which while the overall percentages of benign apps incorrectly rejected is low, the count is still orders of magnitude greater than the count of those correctly rejected.

Experiment design. To demonstrate auPRC's superiority in imbalanced datasets, we look at two different datasets. The 1:1 and 1:100 ratio datasets, one of which is very balanced and the other one is very imbalanced. We note that these two results are related to the experiments of RQ3 examining how ratio imbalance effects results.

	1:1	1:100
TPR	96.1%	96.2%
FPR	5.7%	5.8%
auROC	0.970	0.970
auROC$_{0.01}$	0.098	0.142
auROC$_{0.05}$	0.581	0.604
auROC$_{0.1}$	0.815	0.795
auPRC	0.964	0.456

Table 3: Comparing metrics at two mal. vs. ben. app ratios

Experiment results. We can see from Table 3 that TPR, FPR, and auROC are nearly the same between the two datasets, indicating no degradation due to different malicious to benign app ratios. As advocated by prior researchers [15, 16], we also investigate FPR-bounded ROC curves. In our experiment, we have computed auROC$_x$ for several values of FPR, $x = 0.01, 0.05, 0.1$ as illustrated in Table 3. However, like the regular auROC, we see that the value of auROC$_x$ remains more or less same when we change the ratio of the malicious app set size and benign app set size.

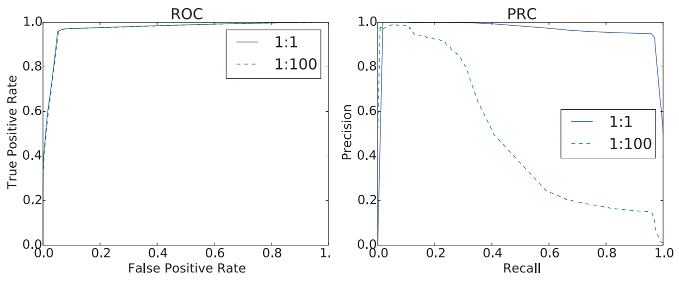

Figure 1: ROC vs. PRC

However, as illustrated in Figure 1, the auPRC is drastically worse with the larger imbalanced dataset. Especially in cases where the optimization and detection of malware is of paramount importance, it is critical that a metric appropriately weighs detection and performance on the minority class as significant. For these reasons, we find auPRC to be a better metric for comparing results of different approaches in ML-based Android malware detection.

4.2 Impact of the input data

RQ2: Does having dated malware in training/test set mislead performance?

The Genome Malware Project [23] for many years has been used as a main source of malware for many ML-based works [4, 5, 6, 9, 21]. However, the Genome set, with malware from 2010-2012, has become a dated source of malware. This RQ seeks to examine if using dated malware sources can lead to misleading results, if used in conjunction with more modern benign datasets as many approaches do.

To answer this RQ, we study the performance of the classifier over two cases: (a) the Genome set is used as the positive class and (b) a subset of modern malware set from our repository (as presented in Section 3.1) is used as the positive class. In both cases, we use a subset of benign apps from our Google Play dataset collected in October 2014.

Experiment design. We utilize a two-class classifier to learn the differences between malicious and benign apps. In Run 1, we use 1,260 Genome malware to represent a dated malware set. For Run 2, we use 1,260 modern malware from VirusShare and Arbor Networks sources. For both runs, the benign set has 63K apps to establish a 1:50 ratio (of malicious to benign apps).

	Genome Malware	Modern Malware
TPR	98.0%	94.3%
FPR	1.8%	7.1%
auPRC	0.601	0.410

Table 4: Genome malware vs. modern malware

Experiment results. As shown in Table 4, we observe that the Genome-based set yields substantially better classification performance when compared to the more modern malware. We note several possible confounding factors that can affect experiments using the Genome apps alongside more modern benign apps.

(i) Changes in Android specifications: Android has quickly evolved in the past 5 years bringing major changes to its APIs, permissions, and intents that have long been used as distinguishing features. Since the Genome dataset was released, Android has gone from API Level 11 to 21. Features whose use has either evolved or devolved over time will look discriminative between malware and benign apps in Run 1, when in reality they have to do with time, not maliciousness. In a sense, in Run 1 the classifier learns to distinguish 2012 from 2014, not malicious from benign apps. Run 2 verifies this by using apps from the same timeframe and shows, lacking the implicit age feature, that performance suffers.

(ii) Lack of diversity of malware in Genome: The Genome dataset only includes around 20 unique malware families, where modern malware are far newer and more diverse. The over triple FPR and almost triple false negative rate (1-TPR) of modern than Genome appears to indicate that the diversity of modern malware makes it substantially more difficult to detect and classify correctly.

We conclude that using the Genome data set should be phased out from modern Android malware detection works due to its ability to mislead performance of systems.

RQ3: Does classifier performance degrade as we approach real-world ratio of malware and benign apps?

Thankfully, the occurence of malware in the app stores is fairly low. However, this imbalance in the amount of malware and benign apps in a store can contribute an interesting factor in classifier performance. In this RQ we experiment with how more realistic ratio imbalances affect classifier performance. Peer works do not consider highly skewed ratios. The highest skewed ratio between benign and malicious apps in the papers we studied was around 1:22 [5] while many others used less skewed ratio than a 1:10 [4, 6, 21].

Experiment design. We begin with a 1:1 ratio represented by 8.5K malicious and 8.5K benign apps. We then add benign apps as necessary to achieve varying ratios up to 1:100. In particular, we experiment with ratios, such as 1:1, 1:5, 1:10, 1:20,1:50, and 1:100. We utilize undersampling to ensure that both classes of sampling are weighted equally when training the classifier.

	1:1	1:5	1:10	1:20	1:50	1:100
TPR	96.1%	95.8%	96.1%	96.0%	96.1%	96.2%
FPR	5.7%	5.9%	5.9%	5.8%	5.8%	5.8%
auPRC	0.964	0.871	0.799	0.701	0.557	0.456

Table 5: Results for different malware to benign app ratios

Experiment results. We can see from Table 5 that performance, in terms of auPRC substantially degrades as ratio increases, although TPR and FPR are more or less constant. We recall that TPR (same as recall) and FPR, and precision are computed at a certain discriminating threshold. One can obtain PRC curves by moving the threshold, thus auPRC is a more comprehensive metric to show a classifier's performance.

RQ4: Does quality of ground truth affect performance?

Peer works generally do some form of ground truth preparation for an ML classifier. Some works [5] require that a minimum of 20% of VirusTotal reports indicate the app is malicious. Other works [21] hold stringent standards and require that the reports return a matching malware family to be used in their dataset. In this RQ we explore how different policies regarding varying levels of certainty of maliciousness affects performance, as well as discussion on the most realistic ways to deal with ground truth preparation.

Experiment design. For both runs, the benign set is 750K benign apps. For Run 1, we randomly select 50K high quality malware (defined in Section 3.1) as the positive class to test against the benign apps. For Run 2, we randomly select 50K low quality malware (defined in Section 3.1) as the positive class and the same benign apps as the negative set.

	High quality malware	Low quality malware
TPR	97.8%	65.0%
FPR	5.1%	28.7%
auPRC	0.846	0.176

Table 6: High quality malware vs. low quality malware

Experiment results. We see from Table 6 that the higher quality malware leads to substantially better performance of the classifier. Note that in the above experiments the tuning parameter is the number of 'positive' flags from AV reports. Several reasons can explain the disparity in performance. High quality malware are likely to exhibit many traditional signs of malware, having certain permissions, APIs, *etc.* These well-defined signs make it easier to detect by AV products. These well-defined attributes also make it perform well in our classifier, since they present an easier problem to the ML algorithm. A 'low quality' malware can be representative of many different things. It could simply be a false positive by an AV product or indeed be a true positive but represent potentially newer and less easily detectable malware, upon which only a few AV products are able to detect. While it would yield better performance to remove low quality malware, for all other RQs we leave low quality malware in. This, then, represents a more realistic base data set, one where the picture is not always black and white and AVs aren't always up to date. We recall that if training and/or test dataset have mislabeled items, that adversely affects the classifier's performance results.

RQ5: Does presence of adware in the dataset affect the performance?

Adware exists in an inevitable grey area between that of malware and benignware. Different works have taken approaches to the placement of adware. Drebin [5] simply drops the adware, while most include them in the malware set. This RQ explores what is an appropriate way to handle adware in evaluating classifier performance.

We report how much impact we see in detection rate when we drop adware from malware set compared to the case when we do not do so (*i.e.*, we include adware in malware set). Finally, we examine the problem from a three-class perspective, and run a 3-class classifier to learn each distinctly.

Experiment design. Run 1: 100K malware only (*i.e.*, no adware) as positive set, and 100K benign apps as negative set. Run 2: 100K malware and 100K adware combined as the positive set, and 200K benign apps as negative set. Note that we increase the benign set to keep a balance between the two-classes. Run 3: a three-class classifier between benign, adware, and malware, with 100K apps of each.

	benign vs. malware	benign vs. malware + adware	three class classifier
TPR	79.6%	80.6%	76.2%
FPR	18.8%	15.7%	11.9%
auPRC	0.843	0.884	0.780

Table 7: Assigning adware to different categories

Experiment results. From Table 7, we observe that when adware and malware are combined together, we see the best performance. We note that the metrics for the three-class classifier are actually the average of all three of the classes specific metric (*e.g.*, auPRC is the average of the auPRC of malware, adware, and benign.) As performance without adware in the mix (Run 1 vs. Run 2) degrades, we hypothesize that adware in a sense is indeed very distinct from benign, thus boosting the performance of the classifier when it is included in malware. Furthermore, we can conclude that malware and adware are quite similar, as we see the worst results when we attempt to distinguish between all three in a three class classifier.

We know, in real world, people seek to separate adware from malware, but our experiment shows that it will be challenging. Further research could seek the best features for discriminating adware. In our opinion, simply dropping adware from the experiment dataset (and evaluating only on malware and benign apps) is not recommended since in the real world all three classes exist.

4.3 Impact of algorithm sophistication

RQ6: Are more features better?

We observe that different ML-approaches utilize different number of features. While working out a ML-approach the designer can be interested to investigate whether all of the items in the feature set are informative, *i.e.*, they are actually helping the classifier in discriminating the classes. It is obvious that there is a downside of choosing features that are not discriminative as they will increase the computation complexity without delivering any benefit. In this RQ, we investigate whether we always get better classification performance while using more features. To get working examples for this study, we emulate two existing approaches' feature sets (Drebin and DroidSIFT), which provide two canonical cases as explained in Section 3.2.

Experiment design. First experiment (with emulated Drebin features). As discussed in Section 3.2, our emulated Drebin feature set contains over 1 million features. We build a condensed subset of features by removing those features that are raw strings appearing in particular apps. In particular, we removed the 'url's, and 'component name's from the total feature set. The reduced feature set contains only 2,246 features, and they correspond to semantic behaviors (as defined by permissions, APIs, *etc.*) as opposed to an arbitrary association, which is the case for 'url' or 'component name' features. In this RQ, we evaluated the classifier with the reduced set of features.

Specifically, we perform two runs by varying the feature set. Run 1: full set of emulated Drebin's features (more than 1 million); Run 2: the reduced set of 2,246 features.

Second experiment (with API-dependency-graph-based features). DroidSIFT uses a unique type of features that are graph distance based where those graphs represent API dependency. This unique type makes us interested to emulate DroidSIFT features to study RQ6. However, we stress that we did not work with the DroiSIFT paper's original dataset nor we emulate their entire approach. In no way, we claim that our results reflect the original DroidSIFT approach's performance. We only emulated their feature vector design (with the help of the DroidSIFT authors) and that too uses a different dataset and a different reference database of graphs as discussed in Section 3.2. To highlight the difference of the emulated features and the original DroidSIFT features, we refer to the emulated feature set as 'API-dependency-graph-based features' from now on.

We recall that our final dataset had 1,524 apps and each app is represented by a feature vector of length 1183 (which is the number of unique graphs generated from the training set of 1128 apps). Each element in the feature vector represents a normalized graph distance. In this RQ, we check how informative these features are. First, we performed a mutual information again analysis, and found that only 192 features provide positive information gain.

In the experiment, we ran the k-NN classifier on two combinations of the API-dependency-graph-based features. Run 1: use all of the 1,183 features; Run 2: use only the informative subset of 192 features (features with mutual information greater than 0). This experiment investigates which combination produces better result.

	Emulated-Drebin all features (1.37 million)	Emulated-Drebin 2,246 features
TPR	98.2%	98.2%
FPR	1.5%	0.1%
auPRC	0.911	0.982
auROC	0.990	0.994

Table 8: Results for emulated Drebin features

	API-dependency-graph-based all features: 1183	Most informative 192 features
TPR	90.6%	95.6%
FPR	18.8%	22.1%
auPRC	0.932	0.955
auROC	0.907	0.937

Table 9: Results for API-dependency-graph-based features

Experiment results. First experiment. As illustrated in Table 8, removing all 'url' and 'component names' features, the performance of the classifier actually increases with auPRC for the malware class going up from 0.911 to 0.981. This indicates that emulated Drebin full feature set has numerous non-informative features. Further, we have found that many of these 'url' or 'component name' features appear in only one or two apps and obviously do not represent true discriminative characteristics.

Second experiment. The results in Table 9 show a similar trend with the ones in the first experiment. Namely, by keeping only the top 192 most informative features, the accuracy of the classifier increases by 2%, with auPRC going from 0.932 to 0.955. This would suggest that some of the features in this dataset act as noise, misleading the classifier.

5. RELATED WORK

The research community has studied ML-approaches for Android security vetting as well as application of ML for the general field of computer security. Below we mention a handful of such research works.

5.1 ML as applied for Android Security

Several ML-inspired solutions [5, 6, 9, 21] are to detect malware apps. We stress that our main contributions in this paper are the identification of challenges of applying ML for Android malware detection while other works focus on designing a specific ML-approach. We discuss them below.

Drebin [5] works with a massive feature set (more than 500K features) containing different types of manifest features (permissions, *etc.*) and 'code' features (URLs, APIs *etc.*) as discussed in Section 3.2.2. Yet, Drebin authors demonstrated that the malware detection system is scalable, and it can even run on a phone in order of seconds. Drebin's performance results (the reported TPR, FPR, and ROC) are also very impressive.

DroidSIFT [21] is unique in designing features in terms of distance among API dependency graphs. It builds the API dependency graphs G for each app, and then constructs the feature vector of the app. The features represent the similarity of the graphs G with a reference database of graphs of known benign apps and malware apps. Finally, the feature vectors are used in anomaly or signature detection.

MAST [9] is a triage architecture whose goal is to spend more resources on apps that have a higher probability of being malicious, thereby reducing the average computation overhead for app vetting. This system utilizes a statistical method called Multiple Correspondence Analysis (MCA). It uses permissions, intents and the presence of native code to determine the probabilities of being malicious.

MUDFLOW [6] argues the pattern of sensitive information flows in a malware is statistically different from those in benign apps, which can be utilized for malware detection. From an app, it extracts the flow paths through static analysis, and these paths are then mapped to a feature vector that is used in a classifier.

Moreover, we are the first to study market-scale apps (nearly 1 million benign apps and 250K problematic apps) in evaluating a ML-approach while for existing works the largest benign set had 120K apps [5] and the largest malware set contained 24K apps [6].

5.2 ML as applied for Computer Security in general

There is a large body of work in the intersection of ML and the general field of computer security. Sommer et al. [16] thoroughly studied the challenges of applying ML in network intrusion detection system (NIDS). They observed that ML approaches face much harder challenges when we apply them for anomaly detection in NIDS compared to ML's applications in other domains, such as recommendation systems, image processing, *etc.*. They explained that the above mainly stems from high price of false alarms in NIDS as they waste (already overburdened) network operators' valuable time. This was our motivation to study the metrics related to bounded ROC curve for RQ1. Another notable work in the intersection of ML and computer security is [15].

6. CONCLUSION AND FUTURE WORK

In this paper, we attempted to make the research community aware of common challenges faced by ML-approaches for Android malware detection. We studied the impact of these challenges on our experimentation framework by varying several parameters that correspond to the above challenges. Our study is expected to encourage the practice of employing a better evaluation strategy and/or a better design of future ML-based Android malware detection systems.

In future work, we will extend the current study. As one direction, we will study the impact of the choice of classifier (*e.g.*, SVM, *k*-NN, *etc.*) on the performance. Furthermore, prior researchers [8, 19] showed that how an attacker might exploit her knowledge (even partial) about the classifier to evade the detection. For instance, knowledge on items, such as the feature set, training dataset, and/or classification algorithm [19] can help the attacker evade. Note that the RQs of the current paper investigate influence of some of these items, but we do not yet consider the attacker's knowledge. We will study the impact of attacker's knowledge in future. As an example, Drebin authors [5] have thought about possibility of mimicry / poisoning attack while Drebin uses hundreds of thousands of features (including component names or urls, which can be arbitrary strings). In RQ6, we have observed that only a core subset of features in Drebin classifier actually influence the detection rate. Similarly, our DroidSIFT experiment shows that only a tiny subset of features actually influence the detection rate. We will study in future whether an attacker can exploit the knowledge about the reduced feature set. For instance, the adversary can first search for those apps (from app markets), which the classifier rates very much benign (with respect to the reduced feature set). Then, the adversary can repackage one of these apps with the malicious module and thus may increase her chance of evading the classifier.

Acknowledgment

We thank Daniel Arp, Hugo Gascon, and Konrad Rieck for providing us with additional information on the Drebin work (see Section 3.2.2). We also express our gratitude to Mu Zhang, Yue Duan, and Heng Yin for helping us extract the DroidSIFT features using their private tool (see Section 3.2.3). Our research is supported by the National Science Foundation under Grant No. 0954138 and 1314925. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Furthermore, some of the computing for this project was performed on the Beocat Research Cluster at Kansas State University, which is funded in part by grants MRI-1126709, CC-NIE-1341026, MRI-1429316, CC-IIE-1440548.

7. REFERENCES

[1] Market Share: Devices, all countries, 4Q14 update. http://www.gartner.com/newsroom/id/2996817.

[2] PCWorld Report: Malware-infected android apps spike in the google play store. http://tinyurl.com/lhu9ope, 2014.

[3] Virus Total. https://www.virustotal.com/, December 2014.

[4] Y. Aafer et al. DroidAPIMiner: Mining api-level features for robust malware detection in android. In *Proc. of SecureComm*, 2013.

[5] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of Android malware in your pocket. In *Proc. of the NDSS*, 2014.

[6] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proc. of the ICSE*, 2015.

[7] S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, 3(3):186–205, 2000.

[8] M. Barreno et al. Can machine learning be secure? In *Proc. of the ASIACCS*, 2006.

[9] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for market-scale mobile malware analysis. In *Proc. of the WiSec*, 2013.

[10] J. Davis and M. Goadrich. The relationship between Precision-Recall and ROC curves. In *Proc. of the ICML*, 2006.

[11] Google. Google Report Android Security 2014 Year in Review. http://tinyurl.com/nh4jbue, 2014.

[12] M. Hall et al. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[13] G. Kelly. Report: 97% of mobile malware is on android. this is the easy way you stay safe. http://tinyurl.com/pb7zf2e, March 2014.

[14] E. Lafortune et al. ProGuard. *http://proguard.sourceforge.net*, 2004.

[15] K. Rieck. Machine learning for application-layer intrusion detection. *Dissertation, Fraunhofer Institute FIRST & TU Berlin*, 2009.

[16] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of the IEEE Symposium on Security and Privacy*, 2010.

[17] Sophia. Security Threat Report 2014: Smarter, Shadier, Stealthier Malware. 2014.

[18] N. Viennot et al. A measurement study of Google Play. In *Proc. of the SIGMETRICS*, 2014.

[19] N. Šrndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proc. of the IEEE Symposium on Security and Privacy*, 2014.

[20] W. Yang et al. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proc. of the ICSE*, 2015.

[21] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In *Proc. of ACM CCS*, 2014.

[22] Y. Zhou et al. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proc. of the NDSS*, 2012.

[23] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. of the IEEE Sec. and Privacy*, 2012.