A Similarity-Based Machine Learning Approach for Detecting Adversarial Android Malware

Doaa Hassan[a], Matthew Might, and Vivek Srikumar
University of Utah

UUCS-14-002

[a]Computers and Systems Department, National Telecommunication Institute, Cairo, Egypt.

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

October 20, 2014

Abstract

We introduce a similarity-based machine learning approach for detecting non-market, adversarial, malicious Android apps. By adversarial, we mean those apps designed to avoid detection. Our approach relies on identifying the Android applications that are similar to an adversarial known Android malware. In our approach, similarity is detected statically by computing the similarity score between two apps based on their methods similarity. The similarity between methods is computed using the normalized compression distance (NCD) in dependence of either **zlib** or **bz2** compressors. The NCD calculates the semantic similarity between pair of methods in two compared apps. The first app is one of the sample apps in the input dataset, while the second app is one of malicious apps stored in a malware database. Later all the computed similarity scores are used as features for training a supervised learning classifier to detect suspicious apps with high similarity score to the malicious ones in the database.

1 Introduction

The security of Android smart-phones has been a worthy challenge due the rapid increase of their usage. A big security challenge is when a malware writer uses code obfuscation techniques to develop a malicious application that steals user private information or causes malicious behavior that damages smart-phone resources. As a countermeasure, there have been several attempts to address this issue by automatic detection of Android malware using different machine learning approaches [4, 43, 40, 35, 42]. The basic idea of such approaches is to statically analyze a large corpus of malicious and benign Android applications and extract syntactic or semantic features of android apps that are used to train a classifier to learn a detection model that distinguish the malicious Android applications from benign ones. However, none of these approaches investigates the detection of malicious apps based on measuring the similarity between two apps including the sample app and a malicious app in a predefined set of android malware stored in a database.

In this paper we present a new similarity-based supervised classification mechanism for classifying non-market, adversarial, malicious Android apps from benign ones based on measuring their semantic similarity to a predefined set of other adversarial malicious apps. Our approach measures the similarity between two apps: a sample app in the dataset and a malicious app in the malware database. The static analysis mechanism is used to extract various similarity scores between each compared two apps. This is done by measuring the semantic similarity between the methods of any two compared apps using the normalized compression distance (NCD) [33] in dependance of either **zlib** or **bz2** compressors. The resulting similarity scores are used later as features for training a supervised learning classifier to learn a detection model that classifies Android apps as malicious or benign. Given a training set of a predefined benign and malicious apps, our approach detects other malicious and benign apps by tacking their similarity scores to a predefined set of malicious Android apps. To our knowledge, our approach is the first dedicated supervised learning mechanism that uses the similarity scores of malicious applications that are computed statically as features for automatic classification of malicious Android applications.

We have evaluated different supervised machine learning classifiers in our approach using a small dataset of non-market Android apps provided by DARPA, including: the Support Vector Machine (SVM) [24], Decision Tree [37, 30] and Random Forest [9]. The experimental results shows that our approach achieves the best overall accuracy equal to 93.3% using the SVM classifiers with **zlib** compressor with a false positive rate of 16% and 0 false negatives.

The structure of this paper is organized as follows: In the next section, we provide a precise definition of adversarial Android malware with some motivating examples. In Section 2 we

give an overview of semantic similarity. In Section 4 we describe our approach and explore its various phases including the similarity detection phase, the similarity features extraction phase and the supervised model learning phase. In Section 5 we present our evaluation results. In Section 6, we state some remarks about our approach, discuss the limitations and suggest the various directions for improvement. In Section 7 we investigate the related work. Finally, in Section 8 we conclude with some directions for future work.

2 Definition of adversarial Android applications

Before introducing our classification approach for detecting adversarial android apps, one requires a precise definition of the term "adversarial Android application". There is some research on adversarial software analysis [48, 46], but no one addressed this issue in Android framework or introduced a precise definitions of adversarial application in the context of Android. The only research attempt that we found to address this issue is presented in [47], but it leaves open the precise definition of "adversarial Android applications". We introduce the following definition of that term:

Definition 2.1 *(Adversarial Android applications) Adversarial Android applications refer to those apps that have an injected malware by an adversary in order to make it hard to detect in the original source code even if human being has access to the source code of the apps.*

Based on the definition stated above, we can deduce that adversarial apps are benign apps with a hardly to detect injected malware. In other words, a classical application is taken from a market and the adversary injects her "evil" code (a hook) in the application and propagates the new application in different markets. Following, we introduce some examples of adversarial android applications in a form of benign android applications from Google Play injected with a unique malicious behavior:

Example1: SMSbot It is an app that allows user to automatically send one or more text messages to a contact in an automatic way. The injected malware in this app allows an adversary to abuse it in order to address all of the user's contacts and alter the message being sent to ensure an infinite cycle of SMS message. The attack is triggered by sending the SMS message to a victim phone running SMSBot. Then all SMSBot users reachable via contacts became flooded with SMS messages.

Example2: FileExplorer It is an app that acts as a file manager that allows user to access, manage files such as documents, pictures, music and videos and folders on the Android device and computers and share them with others. The malicious behavior of this app is triggered by calling a malicious method that makes a zip copy of the last file or directory selected in FileExplorer's directory to a hidden directory which will makes the archive gets larger and cause run out of memory or storage space when the process is repeated.

3 Overview of semantic similarity

The semantic similarity has been used extensively to compare elements of various types for their meaning or semantic content as opposed to syntactical similarity representation [39, 27, 36]. Hence, the semantic similarity measure can play an important role in training a classifier or an intelligent agent that benefits from semantic analysis to mimic human ability to compare elements.

The semantic similarity has been incorporated with static analysis in some research for malware detection [23, 32, 11, 8]. The basic idea of that research is to use the structural features of malware represented by its control flow graph (CFG) to assess the similarity of malware. This has been done by matching CFG-signatures. However, the CFG of malware samples in these approaches can be changed by a number of morphing techniques. Meanwhile, the functionality of malware is kept unchanged. In order to address this limitation, the semantic similarity has been used to compare CFGs of a program against a set of control flow graphs of known malware [10].

The semantic similarity has been also used as a metric by dynamic analysis approaches for detection of malware, where the basic idea is measuring the similarity of malware behaviors [6].

Due the cost and limitation of dynamic analysis in detecting multipath malware [21], our work has been motivated by incorporating the semantic similarity with static analysis to detect adversarial android apps with a high immunity to morphinic techniques to avoid malware detection. This has been done by measuring the similarity between CFGs of app's methods and those of malcious apps. Later the resulting similarity score is used as a feature to train a supervised classifier to identify malicious apps.

4 Proposed approach

The basic architecture of our similarity based classification approach is illustrated in Figure 1. First we build a database that consists of various known malicious Android apps from different categories as representative samples. Then we have used Androguard [18], a static analysis tool for detecting android malware to measure the similarity between each app in our input dataset and each app in the android malware database in order to construct a similarity feature technique. This technique extracts the semantic features of the sample android app in our dataset represented by the control flow graphs of its methods (i.e., method signatures) and computes their similarity scores to those of each app in Android malware database. The total similarity score between methods of each pair of compared apps are used as a metric for detecting similar apps. These similarity scores are converted to feature vectors that are used to train a supervised learning classifier to identify the malicious android apps from benign one.

In summary, our similarity based classification approach aims to build a classifier for detecting malicious android apps using the resulting similarity scores percentage for each sample app as a feature. By this way the classifier will automatically identify the malicious pattern resulting from high similarity score between the sample app in an input dataset and the malicious apps in Android malware database. Thus, in the end the classifier learns the detection model from the input dataset to correctly predict a given class of new apps as a malware or benign.

Our approach is divided into three phases: similarity detection phase, similarity feature extraction phase and finally the supervised model learning phase. Following we describe each phase in more details.

4.1 Similarity detection

Our mechanism for detecting similarity between two apps relies on comparing the methods of both apps to detect the semantic similarities of them. We have used the similarity algorithm presented in [19] to find the similarities of methods between two Android applications. The basic idea of the algorithm is to compare each method of the first application with methods of the second application (unless the two methods are identical) using the Normalized Compression Distance (NCD) [14] parameterized with an appropriate compressor. This is done by converting each method to a formatted string that represents its control flow graph, then computing the NCD between the two strings corresponding to a pair of compared methods. Mathematically the NCD between two strings a and b is pre-

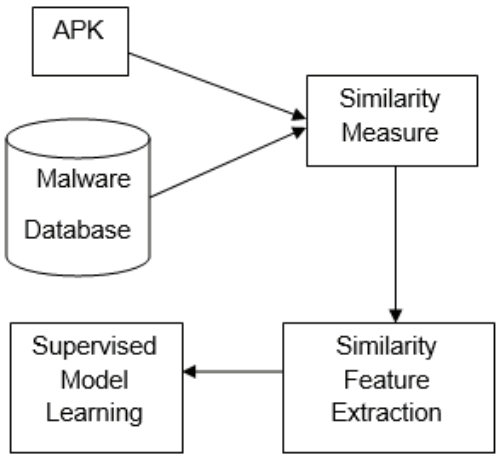

Figure 1: A framework of the proposed approach.

sented as follows:

$$NCD_z(a,b) = \frac{Z(a+b) - min\{Z(a), Z(b)\}}{max\{Z(a), Z(b)\}}$$

where Z(a) refers to the binary length of a string **a** compressed with a compressor **Z** and Z(a+b) refers to the concatenation sequence of two strings **a** and **b** compressed by a compressor Z. If NCD(a, b) = 0, then a and b are similar, while if NCD(a,b) = 1, then they are totaly dissimilar. We summarize the methodology of the algorithm as follows:

- Generating a method signature using the grammar presented by Silvio Cesare in [12]. Using such a grammar each method is converted into a formatted string that represents the control flow graph of the method.
- Identifying all identical methods (i.e., methods that have the same signatures).
- Identify all similar methods (methods that are partially identical) using NCD that represents the information content shared between two sets of methods signatures.
- Calculating the total similarity score between each pair of compared apps according the algorithm presented in [20]. The algorithm applies a designation (between 0.0 for identical methods to 1.0 for totaly different one. In addition to a value of the NCD between 0 and 1 for partial identical methods (i.e., similar methods).

After running the algorithm between two android applications, the following elements are produced and detected:

- Identical methods,
- Similar methods,
- New methods (i.e., methods that do not exist in the first application but they are presented in the second one),
- Deleted methods (i.e, the number of methods that have been deleted in the first application).
- The final percentage score that indicates how much the two apps are similar, more precisely 100% indicates that both apps are identical where lower percentage score indicates that they are too much different.

We have used androsim [3], a python module in Androguard package that implements our similarity algorithm parameterized with different compressors to measure the similarity score between two apps. The final value of similarity score is calculated in our experiment by measuring the NCD between methods of each pair of two compared apps in dependence on two compressors: **zlib** [34], the default compressor in androsim and **bz2** compressor [1]. Then the performance of our approach is compared with both compressors.

4.2 Similarity Feature Extraction

As for the similarity feature extraction phase, we have written a python script that imports the similarity scores resulting from the similarity detection phase. In the end, we retrieved a number of similarity features for each Android app, represented as numeric similarity scores. The values of all features for all sample apps are stored as feature vectors, represented in Attribute-Relation File Format (ARFF) as a list of apps instances sharing a set of attributes. Figure 2 illustrates an example of a feature vector for AgentSmith apk file, one of the sample apps in our input dataset. Each feature vector is labeled as malicious ”M” or non-malicious ”NM” based on a previous feedback that we got from expert static analysts.

We repeated our experiment with a different number of the similarity features as an attempt to refine feature extraction phase and hence improve the performance of model learning phase. This has been done by trying different selections for the number of malicious apps

in our malware database. Each selection is used once in the similarity measure test with each app in the input dataset. Next the performance of the classifier is tested each time according to a specific selection in order to decide the best number of features that achieves the highest accuracy. More details about this stage will be covered in Section 5.

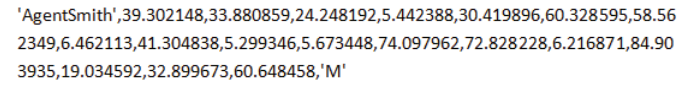

Figure 2: An Example of a feature vector for AgentSmith apk file.

4.3 Supervised Learning with Similarity

During the supervised model learning phase, we feed our similarity scores dataset to various commonly supervised machine learning algorithms in order to build the classification models by learning from them. We evaluate their performance for the classification of unknown apps as either malware or benign. We have generated three different supervised classifiers: Decision Tree [37, 30], Random Forest [9] and Support Vector Machine (SVM) [24]. More precisely the J48, Random-forest, SMO implementations respectively in Weka [25], a free data-mining software. Then we compare the performance/accuracy of those classifiers to the majority classifier, the simplest baseline classifier (ZeroR implementation; the default classifier in Weka) [38]. This is in order to determine the baseline accuracy which simply predicts the majority class as a bench mark for other classifiers. We have used all apps in dataset as training and test data in 10-fold cross validation, a standard approach for evaluating the performance of machine-learning classifiers (where 90% of the dataset is randomly selected for training and the remaining 10% is kept for testing). We have tried different settings for all generated classifiers and pick up the one of highest accuracy. As for SVM, we have evaluated the classifier with different values of penalty constant c and types of kernel, then we chosen the best c and kernel that achieved the highest accuracy. In case of Decision Tree and Random-Forest we evaluated the classifier with different seed number then pick up the best number that leads to the highest accuracy.

4.3.1 Dataset

To extract the similarity features of malware apps as well as evaluating and generating the classification models, we have used a dataset that consists of 39 Android applications and a malware database that consists of 18 malicious Android applications. All of them are non-market application that released as part of the DARPA Automated Program Analysis

for Cybersecurity (APAC) program. The choice of 18 samples in the malware database was decided due to various random selections of that number and pick up the one that leads to highest accuracy of our classification mechanism as it will be explained in the next section. Our dataset consists of 35 malware apps and the remaining four apps are benign. In order to solve the problem of imbllanced dataset (as the number of malware samples is huge in comparison to the number of benign samples), we have injected other 21 benign apps from Google Play Store [2] with different size and categories in our dataset. In the end, the total number of benign apps increased to 25 benign apps against 35 malicious ones.

Although our dataset is small, the malicious behaviors in all apps in our dataset and also in the database of malicious apps are very challenging to detect in comparison with that in other malware apps available on the Android market. It was injected by a third-party within the APAC project that uses an anti-diffing tool on apps to make it hard to detect it in the original source code. This of course makes our dataset are different form other datasets used in the related work for classification based on measuring the similarity. Also this adds a new dimension in detecting the similarity between two apps due to the existence of new maliciousness patterns (indicated in the similarity scores) not produced using commonly used dataset in related similarity detection approaches.

4.3.2 Supervised Learning Models

The generated three classifiers that we have used in our experiment belong to two different family of classifiers. J45 and RandomForest are related to decision trees and SMO belong to function classifiers. We have chosen different machine learning algorithms from different categories in order to build different classifications models for detecting malicious android apps and choose those of a high overall accuracy. The performance of each classifier is evaluated by measuring its mean accuracy and estimating the standard deviation around it as well as the false positive rate (FPR) expressed by the percentage of misclassified benign apps instances as a malware.

The mean accuracy is calculated using the following equation:

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP}$$

where

- TP is the number of correctly classified malicious apps.

- FN is the number of incorrectly classified malicious apps.
- TN is the number of correctly classified benign apps.
- FP is the number of incorrectly classified benign apps.

5 Evaluation

5.1 Experimental setup

For each app in our dataset, our system extracts the semantics features represented by the percentage similarity scores to malcious apps samples in malware database and presents them in a vector. In total, our system produces 60 feature vectors. In order to investigate the influence of the compressor chosen used when calculating NCD between a pair of methods signatures in two compared apps, we test our similarity classification approach when a two standard build in **zlib** or **bz2** compressors for androsim [3] are used to compute the NCD.

As for the malware database, we select randomly various samples of malicious apps that provided by DARPA. We start by picking up randomly five times one malicious sample and store it in the database, then measure the similarity score between each app in the dataset and that app. Then we repeated the experiment several times with increasing the number of malware stored in database gradually and measuring the similarity scores between each app in the dataset and those in the database each time. Our results show that our approach detects the malicious apps with a high overall accuracy of %93.3 using SVM classifier and a malware database that consists of 18 malware samples of different categories in case of zlib compressor. Figures 3, 4 shows the accuracy versus number of features in case of zlib and bz2 respectively. We should mention that the number of apps in the malware database is not the only factor that affects the accuracy, but also the type of malware samples itself and how much the apps in our input dataset are similar to that app expressed by the similarity score value as we will highlight more in the next section.

We have run our approach for detecting similarity on a standlone Mac desktop machine with 2.8 GHZ processor Intel core 17 and 16 G Memory Rams. We notice that the processing time of measuring the similarity between two apps varies due to the size of the tested app. Figure 5 illustrates the app size versus its processing time similarity measure.

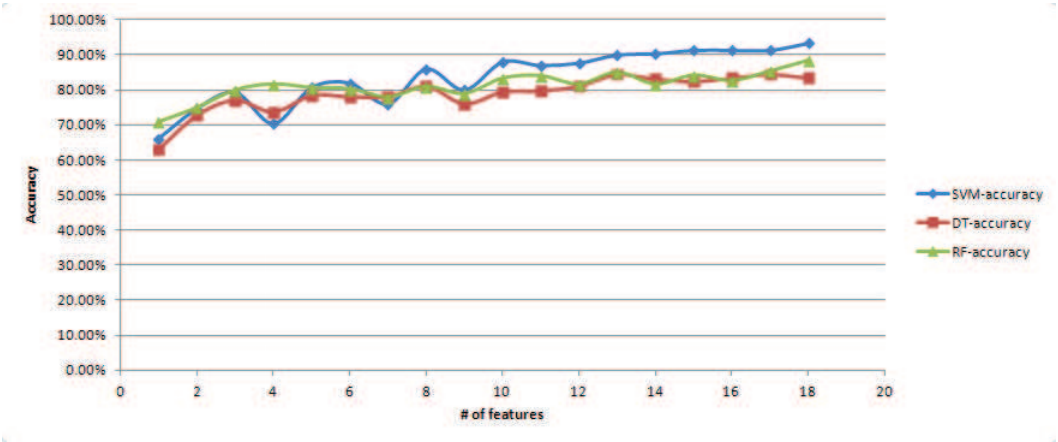

Figure 3: Number of features versus accuracy (zlib compressor).

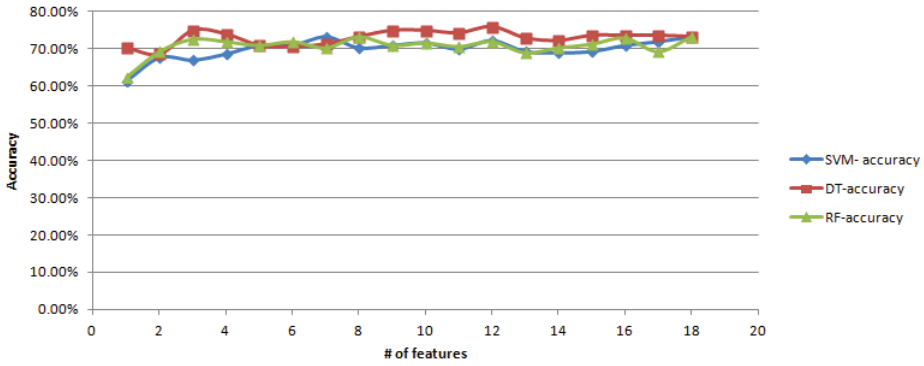

Figure 4: Number of features versus accuracy (bz2 compressor).

5.2 Results

We have run and analyze our experiment in Weka Experimenter Environment [41]. We have run all the generated classifiers 10 times with different settings of C constant in case of SVM (c= 0.0625, 0.125, 0.25,.., 16, 32) and seed number (s= 1, 2, 3,....., 10) in case of Decision-Trees and Random-Forest. The classification results with a similarity measure parameterized by **zlib** and **bz2** compressors are shown in Table 1. The classification schemes used in the experiment are shown in rows. The first row refers to the baseline classifier (i.e., the classifier that predicts the majority). For each classifier we have reported about its performance by the accuracy, the standard deviation and the false positive ratio as we mentioned in Section 4.3. As shown from Table 1, the classification results of the three generated classifiers are statistically better than majority classifier at a significance level equal to 0.05 according to paired t-test [44]. Also we notice from the table that we got the highest accuracy (93.3%) and lowest false positive ratio (16%) using SVM classification (with C=32) with a similarity measured using zlib compressor.

	Classification with Zlib compressor			Classification with bz2 compressor		
Ml Algorithm	Accuracy	Standard deviation	FPR	Accuracy	Standard deviation	FPR
Majority classifier	58.33%	8.38%	100%	61.67%	7.68%	100%
SVM	93.33%	9.48%	16%	73.33 %	16.33%	39.1%
Decision Tree	83.33%	15.53%	28%	73.33%	14.15	52.2 %
Random Forest	90%	13.16%	16%	76.67 %	17.1%	39.1%

Table 1: Performance evaluation of our approach in case of zlib and bz2 compressors.

The probability of a false positive in our classification mechanism results from the fact that the similarity measure between two methods relies on measuring the NCD between their signatures that represent their CFGs. Since the CFG analysis is flow-insensitive, meaning all branches and loops are ignored. This may lead to an unsound representation of the two method structures in comparison and which affects the value of NCD that measures the similarity between both.

5.3 Comparison with baseline similarity approach

In order to show the effectiveness of our approach for detecting malicious apps, we have re-trained our classifiers using the simple value of NCD between each compared two apps extracted as a feature and computed in dependence of zlib and bz2 compressors. Mathematically the simple NCD between two apps X and Y is presented as follows:

$$NCD_z(x,y) = \frac{Z(x+y) - min\{Z(x), Z(y)\}}{max\{Z(x), Z(y)\}}$$

where x and y are the binary lengths of X and Y apk files respectively compressed with compressor Z. Z(x+y) refers to the resulting compression of the two apk files x and y.

We call the resulting NCD values the simple similarity metric. We compare the accuracy results that we got using our approach to the one that uses the simple similarity metric as a feature. Tables 2 shows the performance of each classifier on detection of malicious Android applications, measured using the simple similarity metric computed using **zlib** and **bz2** compressors. By comparing the classification results in Tables 1 and 2, clearly our approach outperforms the simple similarity metric classification mechanism. The reason for that is the later does not rely on the semantics features of each app (represented in the signature of its methods) included in the pair-wise comparison like ours.

	Classification with Zlib compressor			Classification with bz2 compressor		
Ml Algorithm	Accuracy	Standard deviation	FPR	Accuracy	Standard deviation	FPR
Majority classifier	63.33%	6.70%	100%	64.33%	5.20%	100%
SVM	69.11	11.93	81.8%	64.33%	5.20%	100%
Decision Tree	63.33	6.7	100%	64.33%	5.20%	100%
Random Forest	51.33%	17.36	81.8%	64.33	5.20%	100%

Table 2: Performance evaluation of simple similarity-based learning models in case of zlib and bz2 compressors.

6 Discussion

In this section we state some remarks about the similarity measure that we use it in our approach and our classification mechanism that use it for detecting malicious apps. Also we discuss the main limitation of our approach and suggest some improvement.

First we found that the similarity measure that we use as feature for training our classifiers is not symmetric. This means that the similarity score resulting from measuring the similarity between app A and app B does not have to be equal to the score of measuring the similarity between app B and A. This of course intuitive since both apps might differ in the number of new/deleted methods even if they have the same number of identical and deleted methods.

Second, We found that the number of apps in the malware database is not the only factor that affects the accuracy of classifiers, but also the type of malware sample itself and how much the malicious and benign apps in our input dataset are similar to that app expressed by the similarity score value. For example using a malware database that consists of 17 malicious samples instead of 18, we got the same accuracy of 93.3% using SVM classifier. However this accuracy decreases to 91.7% and 83.3% if we drop either MyDarawC or SysMon apps respectively from the malware database that consists of 18 malicious samples and re-evaluate the classifier with the resulting 17 malware samples in the database. This due to the high similarity between malicious apps in our dataset and both apps with average similarity score equal to 84.62% in case of MyDrawC and 72.67% in case of SysMon. Meanwhile, the similarity scores between the benign apps in our dataset and both apps are low in comparison with the similarity scores between malicious apps and both apps (65.33% in case of MyDrawC and 54.79% in case of SysMon.)

Third, our results that show that the performance of our classification mechanism with zlib outperforms the one with bz2 coincides with the concluding remarks about the performance of classification with both compressors in other approaches that used NCD for measuring the similarity for different purpose such as the work in [29]. In that work, NCD was used to measure the similarity between the gene sequences for the hemagglutinin of influenza viruses. It was found that the performance of clustering the hemagglutinin (HA) sequences of influenza virus data for the HA gene with NCD using zlib compressor outperforms the one with bz2.

Another aspect of our similarity learning approach is the processing time. We have measured the processing time of computing the similarity score between a number of randomly selected apps in our dataset of various size and three malware samples of different size (namely, AWeather (41KB), FileExplorer(233KB), SMSBot (4.6 MB)). We found that the size affects too much on processing time, but meanwhile it is not only the only factor that affects, but the processing time is also affected by the number of detected identical, similar, new and deleted methods between each two compared apps using the similarity algorithm introduced in Section 4.1. Figure 5 shows a plotting of processing time of measuring the similarity versus app size.

We also found that the processing time is almost the same when measuring the similarity between the randomly selected apps in our dataset and two highly similar apps

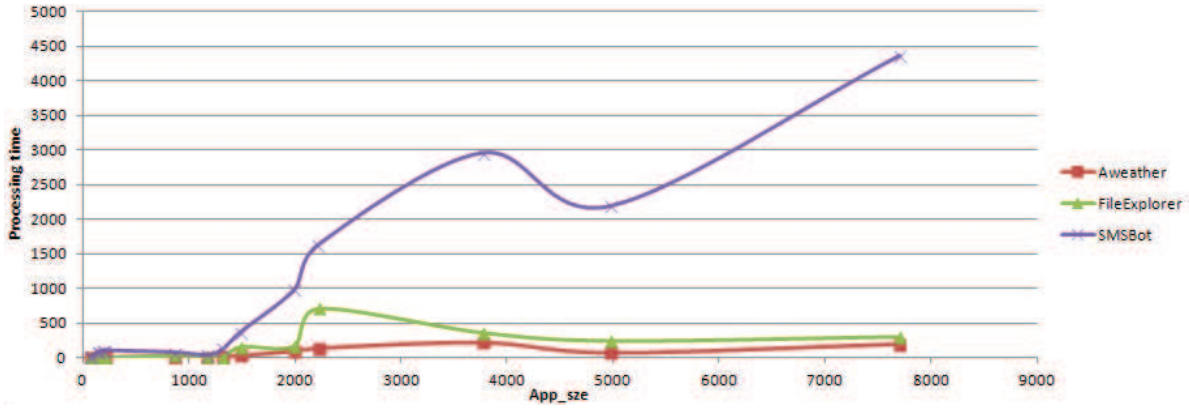

Figure 5: Processing time of detecting similarity versus app size.

(e.g., MyDrawA and MyDrawC in malware database) with almost the same size (MyDrawA=19.03KB, MyDrawC=19.214KB), where both apps have the same number of identical/ similar methods and differ only in the number of new/deleted methods. This is clearly illustrated in figure 6 which shows the processing time of measuring the similarity to MyDrawA and MyDrawC apps respectively. Clearly the similarity measure takes almost the same time in both cases.

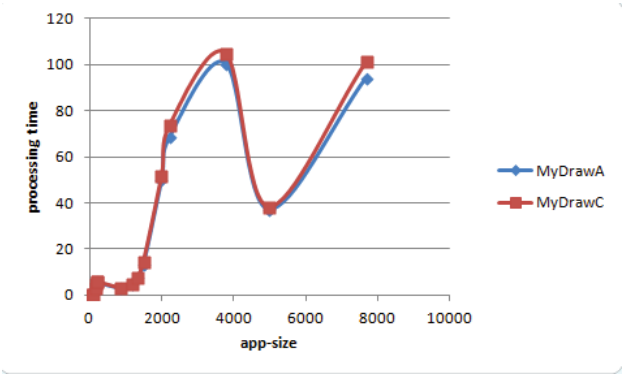

Figure 6: Processing time of detecting similarity versus app size.

Moreover, we have noticed that the processing time of measuring the similarity using NCD in dependence of zlib compressor is less than the one in case of bz2 compressor in most cases. Figure 7 shows the processing-time of measuring the similarity between randomly selected 15 apps from input dataset and random selected eight apps with different size from malware database in case of zlib and bz2 compressors.

Finally, we noticed also that the processing time of the similarity measure is still high, especially with large size apps. Of course this limits the salability of our approach in compar-

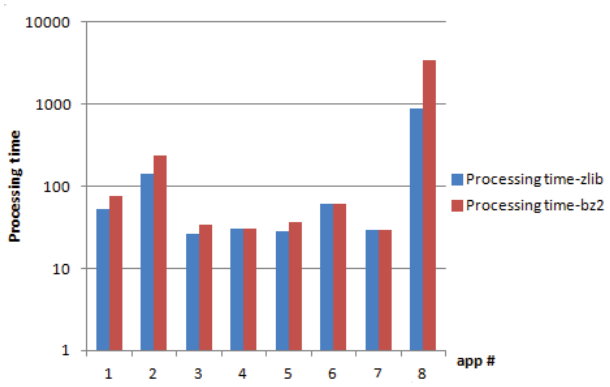

Figure 7: Processing time in case of zlib and bz2 compressors.

ison with other techniques for detecting Android malware based on the similarity measure [50, 15, 17, 26, 16, 22]. However currently our approach is not meant for using to detect malicious android apps in android market, where there are a huge number of these apps. It is meant for detecting a small number of non-markert adversarial apps with a unique malicious behavior. Thus, there is no need to handle the similarity measure for a large number of those apps or accommodate a large number of them as what always done when detecting Piggybacked applications in existing Android Market [50]. Currently, we address the scalability issue of our approach by working on methods of Android apps, and using our mechanism to identify malicious methods instead of malicious apps. This will provide us with large dataset that consists of a large number of malicious and benign methods. We expect such a dataset will improve the accuracy of our classification mechanism.

7 Related work

Learning with similarity for X86-malware detection The detection of malicious x-86 malware by classification/learning using some similarity measurements as features has been extensively investigated in research [28, 31, 49, 7, 5, 13]. For instance, the notion of similarity of binaries based on provenance has been presented in [13]. It states that two binaries are similar if they are compiled from the same source with same compiler. Such notion was used to evaluate a variety supervised classifiers to identify malware. However the similarity in that work was used as labeling scheme which is different from our approach and other similarity based learning approaches for detecting malicious Android apps that use similarity as a feature. Also the notion of similarity of binaries based on provenance does not fit well to find similarity between various malware written by different writers.

M. Bailey et al. [7] used the NCD to measure the similarity of dynamic behavior of Internet malware. Such a measure was used to cluster similar malwares that have a similar behavior into groups. Although they used NCD to measure similarity, their approach is mainly relies on dynamic analysis, where the behavior of the malware is expressed by non-transient state changes that the malware causes on the system. This is different from our approach which relies on static analysis to find similar methods to those in malicious apps in order to express the malicious score of each sample app. Also they use clustering for categorization/ grouping of similar malware behaviors while we used supervised classifiers to detect malicious apps based on the score of their method similarity to those of malicious apps in the malware database.

Learning with similarity for Android-malware detection Recently there have been various research approaches that investigated the classification of malicious Android apps using some similarity measurements as features. Non of them used NCD to measure the similarity between Android applications that can be used later as a metric/feature by the classifier to identify malicious Android applications.

For instance, the similarity between apps has been investigated in [50] using a metric space and proposed linearithmic search algorithm. The algorithm detects piggybacked apps (legitimate apps with attached destructive payload added by malicious authors) using a feature fingerprint technique that extracts various semantics features of apps and converts them to a feature vector. Although this approach is very fast and scalable, it mainly relies on detecting similarity between piggybacked apps and their legitimate versions. It has not been investigated to detect similarity between new malwares derived from different apps and which can be handled using our similarity approach.

The work presented in [15] introduced DNADroid, a tool for detecting cloning/copying of Android applications in order to find variant of the same malware. This is done by computing the similarity between apps based on comparing the program dependency graphs (PDG) between methods in pairs of cloned applications in order to find the semantically similar code at the method level. Although the similarity in that work is detected at the method level like ours, the methodology of computing similarity score of a pair of applications is different as it is based on matching their PDG pairs while ours uses the normal compression distance.

In [17], DEXCD has been presented to detect Android clones between two apps by comparing similarities between all pairings of hashed supsequent pcodes within the methods being compared in both apps.

In [26] a similarity analysis architecture called Juxtap was presented for detecting code reuse in Android applications that indicates piracy as well as if the installed apps are instance of known malware. The architecture of Juxtap consists of four stages: application preprocessing where apk is converted to basic block format, then feature hashing the application to produce feature vector representing the application. In the third stage clustering is used to determine similarity among applications based on calculating a pairwise distance matrix between all applications as an extracted feature. The last stage is the containment analysis which determines the common features between applications and outputs the percentage of common code.

The work presented in [16] investigated the detection of similar apps by different developers (clones) and similar apps from the same developer (rebranded) apps in order to detect new variants of known malware and new malware. It automatically detects the similarity among Android apps using two stages of clustering. In the first stage Locality Sensitive Hashing (LSH) is used to group semantic vectors into features while in the second stage Min-Hash is used to detect fully similar apps and partially similar apps. The similarity detection in that work is determined based on Jaccard similarity coefficient [45].

In comparison with these approaches, Our approach is more effective if malware undergoes a code packing transformation to hide its real content. This is due to using the decompilation technique of structuring for constructing a control flow graph signatures of methods (i.e, method signatures)[12] used by NCD measure.

The most related work to ours is AndroSimilar [22]; a syntactic signature mechanism for finding the statistical similar regions with known malware based on normalized entropy in order to detect the unknown one. Suspicious apps are detected using AndroSimilar by verifying the similarity of their generated signature with existing Android malware signatures in the database. However AndroSimilar is different from our approach in that it considers the syntactic similarity of whole file instead of considering the semantic similarity of all methods like ours, which might lead to inaccurate classification.

8 Conclusions and Future work

In this paper we have presented a new similarity-based supervised machine learning approach for detecting android malware. We have evaluated our approach using different supervised classifiers for predicting whether Android app is malicious or benign. In order to determine the feature vector of the classifiers, we have statically measured the similarity score between each app in our input dataset and each app in the malware database (rep-

resented by the percentage of method similarity between each compared two apps). The similarity scores are used as features to train the classifiers to learn the detection model that identifies the malicious apps.

Our similarity measure relies on the semantic features extracted from the control flow graph of methods in each app. These features are represented in methods signatures, where each signature is represented by a string. The NCD is used to measure the similarity between two strings as an indication to the similarity between two methods. This contributes in the end the percentage of similarity score between two compared apps.

Our classification results shows that the SVM classifiers achieve the best accuracy results. Our work can be extended in various ways: first our classifiers can be evaluated using the similarity measure parameterized with other compressors such (e.g., Snappy, LZMA and XZ) and choose the best performance. Second, we are looking forward to extending our approach to identify malicious methods in android malware rather than classifying malicious apps. This can be done by evaluating our approach on a large dataset of methods collected from different android malware and performing classification based on the similarity edit distance between each method in the input dataset and malicious methods in the malware database. This will aid in capturing the specific part of the code that triggers the malicious behavior in Android malware.

9 Acknowledgments

We thank Marko Dimjaevic, Simone Atzeni, Michael Ballantyne and Amal Youssef for useful discussions. We also thank several members of UComninator group for their proof reading and useful suggestions. This material is based upon work supported by DARPA under agreement number FA8750-12- 2-0106.

References

[1] http://www.bzip.org/.

[2] Google play. https://play.google.com/store.

[3] Similarities/differences of applications. http://code.google.com/p/elsim/wiki/Similarity.

[4] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *SecureComm*, pages 86–103, 2013.

[5] F. H. Abbasi and R. J. Harris. Intrusion detection in honeynets by compression and hashing. In *Telecommunication Networks and Applications Conference (ATNAC), 2010 Australasian*, pages 96–101, Oct 2010.

[6] M. Apel, C. Bockermann, and M. Meier. Measuring similarity of malware behavior. In *In proceedings of the 34th Annual IEEE Conference on Local Computer Networks, LCN 2009, 20-23 October 2009, Zurich, Switzerland*, pages 891–898, 2009.

[7] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *In Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, pages 178–197, 2007.

[8] G. Bonfante, M. Kaczmarek, and J. Marion. An implementation of morphological malware detection. In *EICAR'08*, pages 49–62, 2008.

[9] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[10] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *In proceedings of Third International Conference on Detection of Intrusions and Malware & Vulnerability Assessment DIMVA06, Berlin, Germany, July 13-14*, pages 129–143, 2006.

[11] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security & Privacy*, 5(2):46–54, 2007.

[12] S. Cesare and Y. Xiang. Classification of malware using structured control flow. In *In Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing - Volume 107*, AusPDC '10, pages 61–70, 2010.

[13] S. Chaki, C. Cohen, and A. Gurfinkel. Supervised learning for provenance-similarity of binaries. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 15–23, 2011.

[14] R. Cilibrasi and P. M. B. Vitnyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51:1523–1545, 2005.

[15] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *ESORICS*, pages 37–54, 2012.

[16] J. Crussell, C. Gibler, and H. Chen. Andarwin: Scalable detection of semantically similar android applications. In *In Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, pages 182–199, 2013.

[17] I. Davis. DEXCD. Technical report. http://www.swag.uwaterloo.ca/dexcd/index.html.

[18] A. Desnos. androguard - reverse engineering, malware and goodware analysis of android applications ... and more (ninja !). http://code.google.com/p/androguard/.

[19] A. Desnos. Android: Static analysis using similarity distance. In *In proeedings of 47th Hawaii International Conference on System Sciences*, pages 5394–5403, 2012.

[20] A. Desnos and G. Gueguen. Android: From reversing to decompilation. Technical report. ESIEA: Operational Cryptology and Virology Laboratory.

[21] V. M. Dolly Uppal and V. Verma3. Basic survey on malware analysis, tools and techniques. *International Journal on Computational Sciences & Applications IJCSA*, 4(1), 2014.

[22] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal. Androsimilar: Robust statistical feature signature for android malware detection. In *In Proceedings of the 6th International Conference on Security of Information and Networks*, SIN '13, pages 152–159, 2013.

[23] H. Flake. Structural comparison of executable objects. In *In Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment Workshop DIMVA 2004, Dortmund, Germany, July*, pages 161–173, 2004.

[24] S. R. Gunn. Support vector machines for classification and regression. Technical report, 1998.

[25] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[26] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *In Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 62–81, 2013.

[27] S. Harispe, S. Ranwez, S. Janaqi, and J. Montmain. Semantic measures for the comparison of units of language, concepts or entities from text and knowledge base analysis. *CoRR*, abs/1310.1285, 2013.

[28] K. Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Transactions on Information and System Security*, 6:443–471, 2003.

[29] T. Z. Kimihito Ito1, , and Y. Zhu2. Algorithms and applications. chapter Clustering the Normalized Compression Distance for Influenza Virus Data, pages 130–146. Springer-Verlag, 2010.

[30] C. Kingsford and S. L. Salzberg. What are decision trees? *Nature biotechnology*, 26(9):1011–1013, sep 2008.

[31] J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *In Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'04, pages 470–478, 2004.

[32] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 207–226, 2006.

[33] M. Li, X. Chen, X. Li, B. Ma, and P. M. B.Vitnyi. The similarity metric. In *IEEE Transactions on Information Theory*, volume 5, Decemeber 2004.

[34] J. loup and M. Adler. http://www.zlib.net/.

[35] N. Peiravian and X. Zhu. Machine learning for android malware detection using permission and api calls. In *In Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, ICTAI '13, pages 300–305, 2013.

[36] S. K. Poonam Chahal, Manjeet Singh. An ontology based approach for finding semantic similarity between web documents. *International Journal of Current Engineering and Technology*, 3(5), 2013.

[37] J. R. Quinlan. Decision trees and decision-making. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(2):339–346, 1990.

[38] M. H. R. K. P. R. A. S. D. S. Remco R. Bouckaert, Eibe Frank. Weka manual for version 3-7-8. Technical report, January.

[39] P. Resnik. Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language. *Journal of Arificial Intelligence Research*, 1999.

[40] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *In Proceedings of the 2012 European Intelligence and Security Informatics Conference*, EISIC '12, pages 141–147, 2012.

[41] D. Scuse and P. Reutemann. Weka experimenter tutorial for version 3-5-5. Technical report, January.

[42] A. Shabtai. Malware detection on mobile devices. *In Proceedings of 2013 IEEE 14th International Conference on Mobile Data Management*, 0:289–290, 2010.

[43] A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *In Proceedings of the 2010 International Conference on Computational Intelligence and Security*, CIS '10, pages 329–333, 2010.

[44] R. Shier. Statistics: 1.1 paired t-tests - statstutor. Technical report, 2004. Mathematic Learning Support Center.

[45] N. H. Sulaiman and D. Mohamad. A jaccard-based similarity measure for soft sets. *In Proceedings of 2012 IEEE Symposium on Humanities, Science and Engineering Researchy*, pages 659–663, June 2012.

[46] Y. C. K. Todd McDonald and A. Yasinsac. Software issues in digital forensics. *ACM Operating Systems Review*, 43, 2008.

[47] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Taguen. A5: Automated analysis of adversarial android applications. Technical report, February 2014. Working paper, Technical Report CMU-CyLab-13-009.

[48] A. Walenstein and A. Lakhotia. Adversarial software analysis:challenges and research. Technical report.

[49] S. Wehner. Analyzing worms and network traffic using compression. *J. Comput. Secur*, pages 303–320, 2007.

[50] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "piggybacked" mobile applications. In *In Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 185–196, 2013.