Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/jisa

AndroSimilar: Robust signature for detecting variants of Android malware

Parvez Faruki[*], *Vijay Laxmi, Ammar Bharmal, M.S. Gaur, Vijay Ganmoor*

Computer Engineering Department, Malaviya National Institute of Technology Jaipur, Rajasthan 302017, India

ARTICLE INFO

Article history:
Available online 18 November 2014

Keywords:
Android malware
Code obfuscation
Similarity digest
Statistical features

ABSTRACT

Android Smartphone popularity has increased malware threats forcing security researchers and AntiVirus (AV) industry to carve out smart methods to defend Smartphone against malicious apps. Robust signature based solutions to mitigate threats become necessary to protect the Smartphone and confidential user data. Here we present AndroSimilar, an approach which generates signatures by extracting statistically robust features, to detect malicious Android apps. Proposed method is effective against code obfuscation and repackaging, widely used techniques to propagate unseen variants of known malware by evading AV signatures. AndroSimilar is a syntactic foot-printing mechanism that finds regions of statistical similarity with known malware to detect those unknown, zero day samples. We also show that syntactic similarity considering whole app, rather than just embedded DEX file is more effective, contrary to known fuzzy hashing approach. We also apply clustering algorithm to identify small set of family signatures to reduce overall signature database size. Proposed approach can be refined to deploy as Smartphone AV.

1. Introduction

Penetration of Android Smartphone and Tablets has increased manifold due to ease of use and cheap availability. Android developed for low power, memory constrained embedded devices has captured more than half of the total market share. Gartner's research forecasts sale of more than a billion Android Smartphone devices in 2014–15 (G. Inc, 2013). Android is popular as it is open source, developed by Google and supported by Open Handset Alliance (OHA). Robust application features have increased the popularity of Android. Software distribution on Smartphone platforms differs from conventional Personal Computer (PC). Smartphone applications, popularly called *apps*, are hosted on central repository of Google (G. Play, 2013) (Official Android Market – Google Play), reputed third party app-stores and local third party markets. Apps are installed directly on user's Smartphone unlike download and execute on Windows based Personal Computer systems.

Google Play has nearly a million apps divided into various categories such as Books, Business, Casual, Communication, Security, Media, Lifestyle, Personalization etc. Popular third party repository Amazon (Amazon Inc, 2013) has thousands of in-house free and paid Android apps. Google Play requires Android developers to create an account with a minimal fee. Apps must be self-signed by developers before uploading. This is to ensure trust between the developers and Google. Implicit assumption is that a developer would care for his image once he is certified by his private key. Most of third-party app-

[*] *Corresponding author.*
E-mail addresses: parvezfaruki.kg@gmail.com (P. Faruki), vlaxmi@mnit.ac.in (V. Laxmi), bharmal.ammar@gmail.com (A. Bharmal), gaurms@gmail.com (M.S. Gaur), vijay.ganmoor@gmail.com (V. Ganmoor).

http://dx.doi.org/10.1016/j.jisa.2014.10.011

stores work the same way. Online repositories are streamlined with different protection methods to guard against threats, for example, Google Play has a service called Bouncer to filter malicious apps before uploading them. Bouncer checks the behavior of apps using static as well as dynamic analysis approaches (Oberhide, 2012). Developers may also upload same app(s) on third-party app-stores to strengthen their product reach and earn advertisement revenues. Some apps are available at the Google Play while some are available only with third party markets due to their localized use.

In this research, we present a robust statistical approach that explores robust byte features (Roussev) for capturing code homogeneity among variants of known malware families, which share common attributes. Unknown variants generated with obfuscation techniques are marginally dissimilar but they still defeat AV signature detection approach. Code homogeneity is captured by searching for strong features that are visible within related files, but has a very low probability of occurrence among unrelated files. To detect the similarity of unknown samples with existing malicious ones, we propose AndroSimilar, a byte-level approach which generates variable length signatures to detect unseen, zero day samples crafted out of known malware. AndroSimilar seems effective compared to Xiang et al. (Zhou et al., 2012a) approach to detect repackaged apps on third party Android markets based on Context Triggered Piecewise Hashing (CTPH) technique SSDEEP (Kornblum, 2006), which considers only Dalvik-opcodes within DEX files embedded in apps.

Rest of the paper is organized as follows. In Section 2 we briefly discuss Android APK file format with a discussion on reverse engineering, repackaging and its implication, and associated tools. Motivation and contributions are also covered in the same section. AndroSimilar approach is covered in Section 3. Clustering Algorithm to reduce the signature set family-wise is explained in Section 4. Evaluation of proposed approach is covered in Section 5, followed by discussion in Section 6. Related work is discussed in Section 7. This is followed by conclusions and future work in Section 8.

2. Background

Increasing popularity of Android OS has attracted malware developers to exploit Android ecosystem by inserting code obfuscation and/or repackaging popular paid apps to distribute them in less monitored third party markets (Castillo, 2012). Repackaging is a process of inserting additional payload, possibly malicious, inside a benign app using reverse engineering techniques. Malware authors use available tools to dissect popular apps, insert malicious payloads like Adware, Spyware, IMEI/IMSI stealer, Bots, Premium number subscription based SMS Trojans etc., repackage and distributed modified apps through less monitored third party markets. Boxer SMS Trojan discussed in Andre and Ramos (2013) employed this approach to extract money by inserting malicious logic in Need For Speed Shift, a popular app on official market. Repackaging also hurts the original app developer reputation and revenues, as malware writers may replace publisher identification of affiliated ad networks with their own to earn money. Repackaged apps pose a serious threat to the centralized app distribution system. Users trust Android app markets assuming that installed apps do not contain illegitimate contents to spy on their activities or steal user information or drain their money. Slow response from OEMs and carriers for releasing frequent updates and security patches increases the lifetime of exploits used by repackaged malicious apps. Conventional signature based approach has worked effectively to prevent known malware families, considering limited capability environment within Android embedded-devices. But nevertheless, it is not robust against zero-day variants of known malware families.

2.1. *Android app structure*

Android app is written in Java and native programming using C/C++ is also supported. Java code is compiled as class files (.class) and then converted into single storage-efficient Dalvik Executable (DEX) file using *Dx* tool (A. Inc., 2013). DEX bytecode executes on Dalvik Virtual Machine (DVM), a register based Virtual Machine specifically implemented for limited resource environment, unlike stack-based Java Virtual Machine (JVM). Typical app structure is shown in Fig. 1.

AndroidManifest.xml has important metadata such as Package Name, Permissions required, definition of various components such as Activities, Services, Receivers and Content Providers. *Resources* folder consists of UI-layout, icons, images, string/color/dimension constants, animation files etc. compiled into binary format as well as other user-defined files. *classes.dex* file contains Dalvik executable code. *META.INF* folder contains digital signature of an app used for verification as well as developer certificate used for identification. This content is compiled into a single package called Android PacKage (APK). Developer self signs it with his private key and publishes it on Google Play or third-party app-stores. Application development is made easier by eclipse environment with ADT plugin (Zheng et al., 2012).

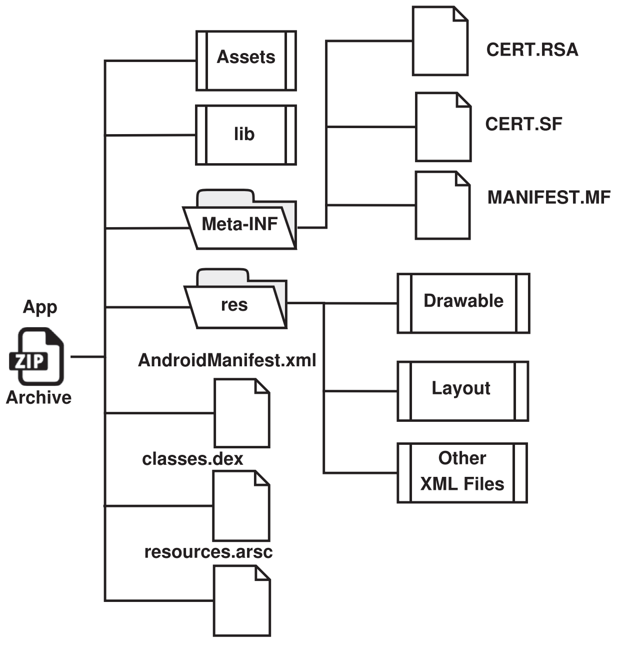


Fig. 1 – APK File format.

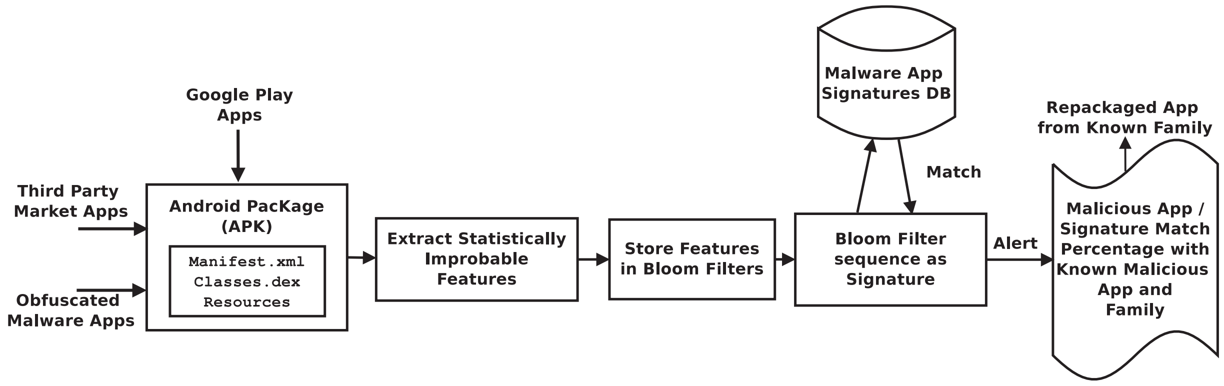

Google Play Apps
Third Party Market Apps
Obfuscated Malware Apps
Android PacKage (APK)
Manifest.xml
Classes.dex
Resources
Extract Statistically Improbable Features
Store Features in Bloom Filters
Bloom Filter sequence as Signature
Malware App Signatures DB
Match
Alert
Malicious App / Signature Match Percentage with Known Malicious App and Family
Repackaged App from Known Family

Fig. 2 – Proposed approach.

2.2. *App reverse engineering and repackaging*

Android apps are more amenable to decompilation in Java source compared to structural language such as C. Apps are reverse engineered to debug them or to perform obfuscation or repackaging on them. Reverse engineering steps generally start with converting Dalvik bytecode into Java bytecode using tools such as *dex2jar* (Dex2Jar, 2013). Then Java bytecode is converted to Java source using decompilers such as *JAD* (JAD, 2013). Obfuscation of apps using tools like *ProGuard* (ProGuard, 2013) makes it much harder to obtain its Java source.

apktool (APKTool, 2013) is one example of reverse engineering and repackaging tool. It disassembles DEX bytecode using *smali-baksmali* (BakSmali, 2013) disassembler into a readable format loosely based on Jasmin bytecode language. After making some changes in it, same can be repackaged using *apktool* again. This repackaged app is then self-signed with another certificate and uploaded to the app-stores.

2.3. *Motivation and contributions*

Current Anti-Malware apps on Android rely mainly on signature based approaches. Due to limited capability of an app within Android, Anti-Malware solutions cannot be effective and alternative methods are needed to counter zero-day malware. Consider the malware samples detected as a part of Malware Genome Project (Yajin Zhou, 2013). It consists of total 1242 samples spanned into 49 different families, with DroidKungFu3 containing highest 309 apps. Since, a number of sophisticated Android malware has been detected by various Anti-Malware solutions (L. Inc., 2012; Castillo, 2012). Given this fact of rising malware, manual analysis of individual sample or a family is in-feasible. It is of utmost importance to find new, automated methods for signature generation as well efficient detection. We have investigated Similarity Digest Hash (SDHash) (Roussev) approach, used in forensics, to detect variants of malicious Android apps as there is a high probability of similarity among apps belonging to same malicious family.

Our contributions in this paper are as follows:

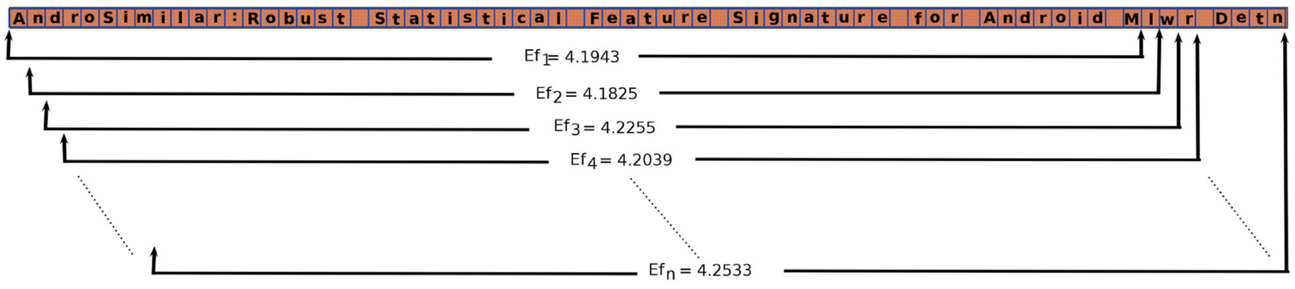
- We have implemented an approach called *AndroSimilar*, which automatically generates signatures for apps, in such a way that it can detect variants of known malware families as well as obfuscated samples of known malicious apps. We have observed empirical distribution and results for both 64 and 128 byte-length features (explained later). We validate the suggestion of Roussev (2009) that 64-byte length features gives reasonable performance and detection rate.
- To make detection more efficient, we have proposed and implemented a signature-set reduction algorithm to determine smaller family-wise signatures.
- We have also analyzed AndroSimilar on just *classes.dex* (DEX) files rather than whole apps because this is the main file which contains executable bytecode. We find that considering just DEX files gives very poor detection rate when they are obfuscated. It also gives inferior detection rate for finding repackaged variants, contrary to fuzzy-hashing based approach (Zhou et al., 2012a) used to detect repackaged apps.

3. Proposed methodology

AndroSimilar approach is shown in Fig. 2, whereas its algorithm is listed in Algorithm 1. We generate signatures of known malware families as our representative database. If similarity score of an unknown app with any existing family signatures matches beyond a threshold, then it is labeled as malicious.

Completely unrelated files should have lower probability of having common features. When two unrelated files share some features, such features should be considered weak as using these shall lead to false positives (Roussev, 2009). Fixed-size byte-sequence features are extracted based on empirical probability of occurrence of their entropy values, then popular features are searched among them according to rarity in neighborhood (Roussev). Diagrammatic representation proposed approach is discussed below.

- Submit Google Play, third-party or an obfuscated malicious app as input to AndroSimilar.
- Generate entropy values for every byte-sequence of fixed size in a file and normalize these in range of [0,1000].
- Select statistically robust features according to similarity digest scheme as representative to the app.


AndroSimilar:Robust Statistical Feature Signature for Android Mlwr Detn
Ef1= 4.1943
Ef2= 4.1825
Ef3= 4.2255
Ef4= 4.2039
Efn= 4.2533

Entropy Calculation:

Feature size chosen is 64 bytes

Possible characters using a byte = 2^8 = 256 characters (i.e., 00000000 to 11111111)

Entropy of first feature = $Ef_1 = \sum_{i=0}^{255}$ **Pi log Pi** where **Pi** refers to probability of occurence of **i**th character(among 256) in feature

$$= \underbrace{7/64*\log(7/64)}_{\text{(7 white spaces)}} + \underbrace{1/64*\log(1/64)}_{\text{(1 colon)}} + \underbrace{2/64*\log(2/64)}_{\text{(2 A's)}} + \underbrace{1/64*\log(1/64)}_{\text{(1 F)}} + \underbrace{1/64*\log(1/64)}_{\text{(1 M)}} + \underbrace{\dots\dots\dots\ 256\ \text{summations}}_{\text{(Many of them would be zeros)}}$$

$$= 4.1943$$

Similarly, entropy of other features is calculated where adjacent features differing by one byte

Fig. 3 – An example: entropy calculation.

- Store extracted features into Bloom Filters. Sequence of Bloom Filters is a signature of an app.
- Compare the signature with the database to detect match with known malware family. If similarity score is beyond a given threshold, mark it as malicious (or repackaged) sample.

3.1. *Normalized entropy values*

Entropy is a useful tool for extracting robust features (Roussev, 2009). Entropy of a given byte-sequence is given as:

$$H(S) = \sum_{i=0}^{255} P(X_i) \cdot \left(\frac{1}{\log(P(X_i))}\right)$$

where $(X_1 \dots X_{255})$ are ASCII characters. It is then normalized into a range 0–1000 using:

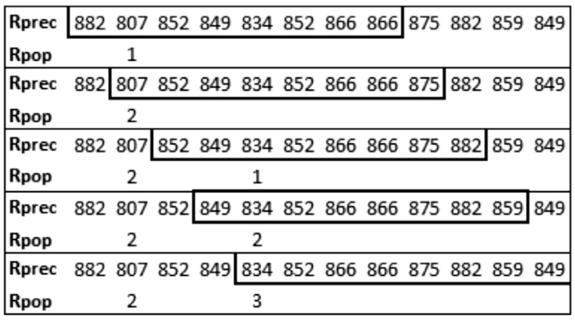
$$H_{normal} = \left\lfloor 1000 \cdot \frac{H(S)}{\log_2 B} \right\rfloor$$

where B is the size of byte-sequence. Fig. 3 shows an example of entropy calculation for $B = 64$. Successive values are computed over sequences shifted by one byte as shown in Fig. 3. Owing to only one byte change, entropy values of neighboring byte-sequences have lower variations. Normalizing entropy values into a much larger range, [0,1000], facilitates comparison. When entropy is measured for byte-sequences of smaller length, variations in entropy value is not large. This leads to features being weak or ambiguous, also processing time is higher (Roussev, 2009). Contrary to that, if byte-sequences of higher length is considered, number of features will go down.

For a given byte-sequence spanning an app, normalized entropy stream is generated in sliding window fashion. Sliding window allows us to calculate normalized entropy of a byte-sequence using the value of previous byte-sequence in a much faster way. As we will see, some app content may be filled with repeated characters. Such attributes create false positives, but they only cover very limited portion of the app. Removing ambiguous features helps reducing false positive rates and it does not have an effect on feature selection.

3.2. *Precedence rank values*

To find least likely features, a Precedence Rank R_{prec} is assigned to each normalized entropy value, such that former is proportional to latter's probability of occurrence (Roussev, 2009). To generate R_{prec} table, we need to analyze entropy distribution of APK files. Figs. 10 and 13 show the normalized entropy distribution, with $B = 64$, of a random set of benign and malicious Android apps respectively. We can see that high-peak at lower entropy values depicts occurrence of same repeated byte-sequences, and essentially weak features. Similarly, peak at higher entropy values depicts compression tables with high randomness, indicative of weak features. Removing such weak



Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		1										
Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		2										
Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		2			1							
Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		2			2							
Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		2			3							

Fig. 4 – Finding popular features.

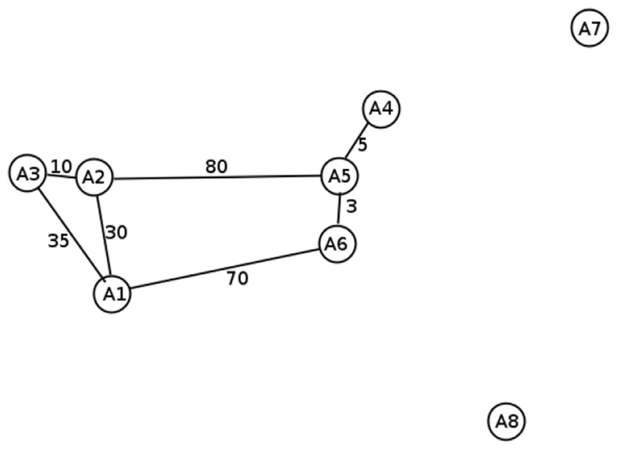


Fig. 5 – Conceptual view of Distance Matrix.

features reduces false-positives with no effect on strong feature selection. We assign R_{prec} to each normalized entropy in the range of 1–1000 by eliminating weak features, where lower R_{prec} means least likely and a discriminant feature. An example of R_{prec} stream is shown in Fig. 4.

3.3. *Statistically robust features*

After generating R_{prec} stream, it is not just important to consider features with lowest ranks. For each R_{prec} value, we maintain a popularity score R_{pop}. We consider a window size of W which runs over R_{prec} stream in a sliding window manner. In each window, left-most minimum R_{prec} is selected and its R_{pop} is incremented. Thus, if R_{pop} of a feature is k, ($k \in [1 \ldots W]$), it means that it is the local minimum within a window of size $W + k - 1$, which further emphasizes relative rarity of that particular feature (Roussev, 2011).

For illustration, consider Fig. 4, where R_{prec} stream of corresponding features of length B-bytes is shown in each row. Window W rolls over consecutive R_{prec} values and left-most lowest one is selected from it. Thus in row 1, R_{pop} of a feature with $R_{prec} = 807$ is incremented and window is shifted to right. Next window (in Row 2), picks up feature with $R_{prec} = 807$ and increments its R_{pop} by one again. Procedure continues till R_{pop} values for all the features are calculated. The features with corresponding R_{pop} values are selected based on a threshold to exclude weak ones. Threshold for R_{pop} is considered such that it enables collection of statistically robust features (Roussev, 2011). For the above example, if minimum popularity threshold is 3, then it considers only the feature with $R_{prec} = 834$ as statistically robust.

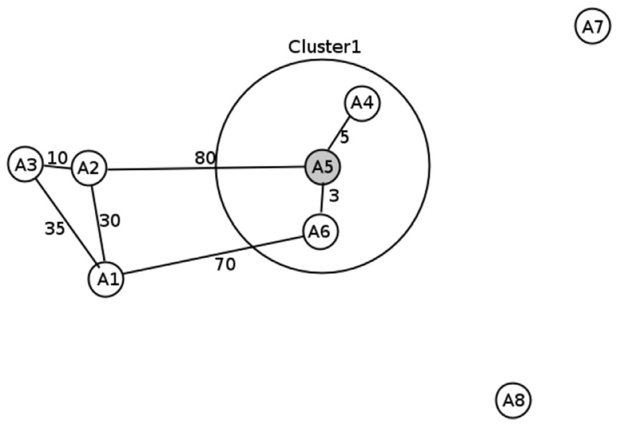


Fig. 6 – Forming cluster with A_5 as representative point.

Algorithm 1. AndroSimilar Algorithm.

```
Input : $mal\_sign\_db$ – Signature database.
        $appset$ – Apps to be checked.
        $threshold$ – Similarity threshold.
Output: $malicious\_set$ – Set of tuples as ($appName$,
        $familyName$).
1  foreach app in the appset do
2      score_results ← an empty array;
3      app_sign ←
       generate_improbable_features_sign(app);
4      foreach malicious_sign in the mal_sign_db do
5          score ← compare_signatures(app_sign,
           malicious_sign);
6          score_results.add(score);
7      end
8      best_similarity_score ←
       find_best_score_match(score_results);
9      if best_similarity_score >= threshold then
10         tuple ← (app.name,
           best_similarity_score.familyName);
11         malicious_set.add(tuple);
12     end
13 end
```

All the robust features selected using above technique are hashed, and then inserted into bloom-filters for compression and efficient searching with limited number of features per single bloom-filter. The sequence of bloom-filters is a signature of an app. Two signatures of corresponding apps are then compared to find a similarity score ranging from 0 to 100 (Roussev, 2011), where lower score means less similarity.

4. Signature-set reduction algorithm

SDHash signature length of an app is on an average 2–3% of its size. Considering the dramatic rise in number of Android malware, instead of maintaining the vast set of these large signatures, only a few signatures can be stored for the purpose, since variants within each malware family are strongly correlated. Hence, we choose to cluster variants (i.e., apps) that are strongly similar and then choose signature of the variant that has many rarest features, discarding the signatures of other family members.

The clustering within each family is done on the basis of SDHash similarity, where the distances among the signatures within a cluster are quite low (i.e., high similarity) according to an empirically chosen *inter-app similarity threshold*. From each cluster, we choose a single point capable of representing all the

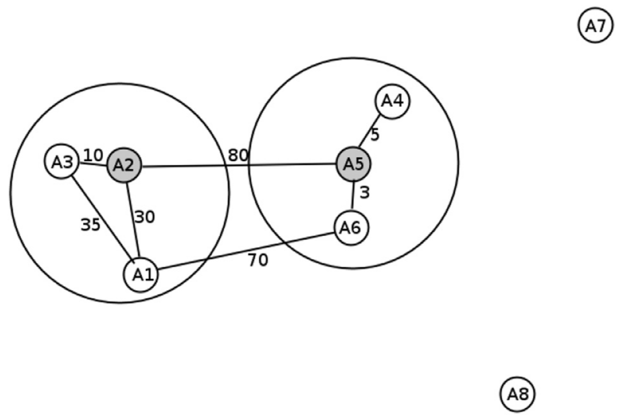


Fig. 7 – Forming cluster with A_2 as representative point.

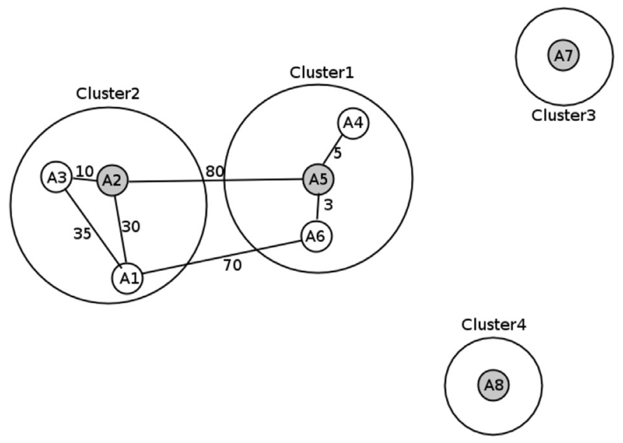
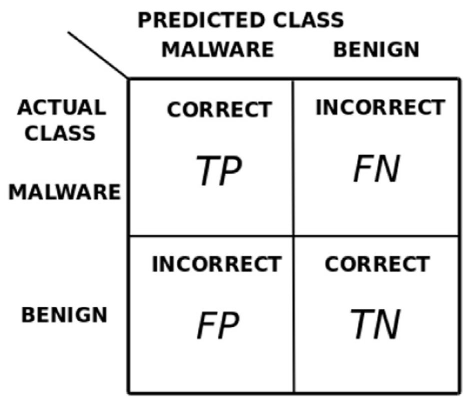


Fig. 8 – Total clusters created after completion of algorithm.

apps in the cluster. Apps very dissimilar to others are represented as a separate cluster. Algorithm is listed in the Algorithm 2, where outcome will be the reduced set of signatures for a particular malware family. However, it should be noted that due to reduction in number of signatures, the chances of identifying unknown variants also decreases.



ACTUAL CLASS \ PREDICTED CLASS	MALWARE	BENIGN
MALWARE	CORRECT TP	INCORRECT FN
BENIGN	INCORRECT FP	CORRECT TN

Fig. 9 – Evaluation model.

Nevertheless, this kind of solution would be helpful when we are dealing with a system under memory constraints such as mobile devices, but maintaining all the signatures would prove worthy as it is better capable of identifying zero day variants.

For the purpose of understanding the algorithm, consider eight apps A_1, A_2, …, A_8 which belong to some particular malicious family. As they belong to same family, assumption is that there would be some extent of similarity between some of them. Say the similarities are of the form as given in Table 1.

4.1. *Simplifying the analysis*

In order to simplify the analysis, lets map these similarities into distances between them. The distance parameter is inversely proportional to similarity among them, we calculate distance simply as:

$$Distance(App_1, App_2) = 100 - SimScore(App_1, App_2)$$

Table 2 depicts the distance matrix among apps. A high value of distance indicates less similarity. A conceptual view of this matrix is shown in Fig. 5.

4.2. *Forming clusters and finding their representative points*

We say two apps are similar only when there is reasonable overlap of common features, i.e., when similarity score of those apps is greater than some similarity threshold, say 35. So the maximum distance D_{max} between apps in a cluster would be $100 - 35 = 65$.

Using distance matrix, we compute ‘neighborhood count’ N_c for each app by counting how many of its neighbors are within D_{max}. In every iteration, we consider an app with maximum neighborhood count $max(N_c)$ as cluster center and its neighbors occupying the same cluster. Thus a large cluster is formed in the first iteration. There might be cases where there would be same neighborhood count for multiple apps. In that case, we calculate total cost of neighbors T_{cost}, which is the sum of distances to its neighbors, and the app having smallest T_{cost} would qualify for cluster center.

Table 3 shows the state after single iteration. As A_5 has highest N_c and smallest T_{cost}, A_5 along with its neighbors A_4 and A_6 qualify to form cluster, with A_5 being representative point of the cluster i.e., its signature would be sufficient to recognize all the apps in the cluster. This is depicted in Fig. 6.

After a cluster is formed, we remove the respective points from the distance matrix also as shown in Table 4.

After second iteration of this process, new cluster A_1, A_2, A_3 is formed with A_2 as representative point as shown in Fig. 7 and corresponding distance matrix in Table 5.

Clustering process is terminated when there are no neighbors of any of the point. Apps that still have an entry in the distance matrix at this point are the ones, which do not belong to any cluster. These apps are placed in new clusters as shown in Fig. 8. We are left with the reduced set of signatures $\{A_2, A_5, A_7, A_8\}$, instead of complete set $\{A_1, \cdots A_8\}$, which could have been used to detect all apps.

Algorithm 2. Signature Reduction Algorithm

```
Input : $mal\_sign\_db$ – Signature database.
        $intra\_fam\_match(App_i)$ – Intra-family signature
        match results for every app-$i$ against other apps
        in the same family.
        $threshold$ – Similarity threshold.
Output: $redu\_sign\_db$ – Reduced signature-set.
1  $mal\_fam\_set$ ← set of all malware families to which
   $intra\_fam\_match(App_i)$ belong;
2  foreach $family$ in $mal\_fam\_set$ do
3      repeat
4          $N_c$ ← An empty array for holding number of
           neighboring apps;
5          $T_{cost}$ ← An empty array for holding sum of
           distances to neighboring apps;
6          $cl\_core\_pts$ ← = Points that are at the cores of
           clusters;
7          $rep\_point$ ← Representative point for the cluster;
8          $new\_cl\_pts$ ← Points of newly formed cluster;
9          $dist\_mat$ ←
           create_distance_matrix($family$);
10         foreach $app$ in $family$ do
11             $N_c[app]$ ←
               find_neighbor_counts($dist\_mat, app$);
12         end
13         if max($N_c$) == 0 then
14             Insert signatures of all apps in $dist\_mat$ to
               $redu\_sign\_db$;
15             break;
16         end
17         foreach $app$ in $fam$ do
18             find_total_costs($dist\_mat, app$);
19         end
20         $cl\_core\_pts$ ←
           find_apps_with_highest_neighbor_count($N_c$);

21         $rep\_point$ ←
           find_apps_with_least_total_count($dist\_mat,
           cl\_core\_pts, threshold$);
22         Insert signature of $rep\_point$ to $redu\_sign\_db$;
23         $new\_cl\_pts$ ← find_neighbors_of($rep\_point,
           threshold$);
24         Append $rep\_point$ to $new\_cl\_pts$;
25         Remove $new\_cl\_pts$ from $dist\_mat$;
26     until $dist\_mat$ is not empty;
27 end
```

5. Evaluation

Fig. 9 displays the evaluation model for the approach. Controlling True Positive (TP) rate and False Positive (FP) rate is very important for any good malware detector. TP means, correct detection of malware, whereas FP means incorrect detection of benign apps as malware. For the purpose of evaluation, we have collected a dataset of 15993 Google Play, 3309 malicious and 5139 third-party apps summarized in Tables 6–8 respectively. Among 3309 malicious apps, 1242 are taken from Malware Genome Project and belong to 49 different families. Rest of them are un-categorized. To test the effectiveness of AndroSimilar thoroughly, we have also generated reasonable number of obfuscated variants of malicious apps using following techniques: 1) Method Renaming, 2) Junk Method Insertion, 3) Change Control Flow Graph and 4) Literal String Encryption.

Our experimental *inter-app similarity threshold* is 25. That means, if the signature of a particular sample matches with some signature in a malicious-set with similarity-score ≥25, that sample is detected as malicious. We shall also discuss the implication of changing this threshold later. In summary, following are the parameter values we considered for experimentation:

- Feature-size B is 64 and 128 bytes.
- R_{prec} tables generated for APKs with B equals to 64 and 128. Also generated for DEX files with B equals to 64.
- Popularity window size W is same as B.
- R_{pop} threshold value is 16.
- Bloom-filter size is 256 bytes.
- Maximum number of features per bloom-filter is 160.
- Inter-app similarity threshold is 25.

5.1. *Byte-sequence length* $B = 64$ *vs.* $B = 128$

Evaluation was done on APK files using two byte-sequence lengths, $B = 64$ and $B = 128$. Recall that smaller value of B requires a lot more processing and generates weak features, whereas much larger value of B will generate less number of features that may not suffice for effective signatures (Roussev, 2009). We have not considered values of B less than 64 for the same reason. Roussev (2009) has suggested that based on practical considerations, such as typical network-packet length, value of B should be 64 or 128. Our evaluation suggests that AndroSimilar with $B = 64$ performs better than $B = 128$. Normalized entropy distribution for $B = 64$ of both benign and malicious APKs can be seen in Figs. 10 and 13 respectively. In the same way, R_{prec} table for $B = 128$ was generated by considering both observations shown in Figs. 11 and 14.

5.1.1. *Detecting unknown variants*

We divided our dataset of 3309 malicious APKs into two sets, first-one with 2854 APKs as a signature-set and second-one with 455 APKs as unknown variants of malicious apps. Result is shown in Table 9, where first row represents unknown variants with fourth and fifth column showing the detection rate for $B = 64$ and $B = 128$ respectively. TP rate of the former is 76.48%, which is almost 3% more than the latter (Tables 10 and 11).

5.1.2. *Detecting obfuscated malware*

We also created obfuscated samples from malicious APK dataset randomly, using four methods as explained before. Obfuscation of an app changes its signature, which helps it to evade AV signatures, if they are based on cryptographic hashes. Result is shown in Table 9, in which rows including and after the third-one shows the detection rate for each obfuscation types separately, for both $B = 64$ and $B = 128$ in the fourth and fifth column respectively. Overall TP rate of the former is 80.65%, which is almost 4% more than the latter.

5.2. *DEX files vs. whole APKs*

classes.dex is the main file that contains Dalvik executable bytecode. It is also very small in size compared to whole app. If

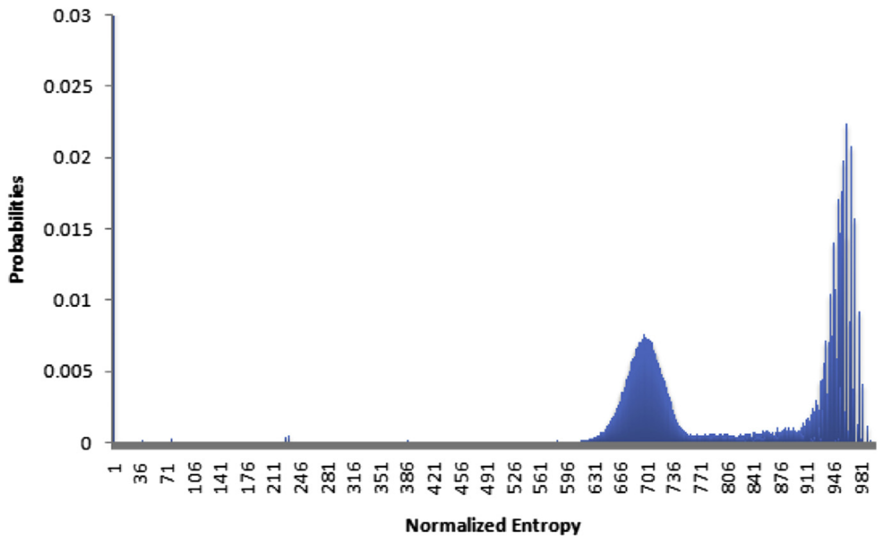


Fig. 10 – Entropy Distribution of 1500 Benign APK Files with $B = 64$.

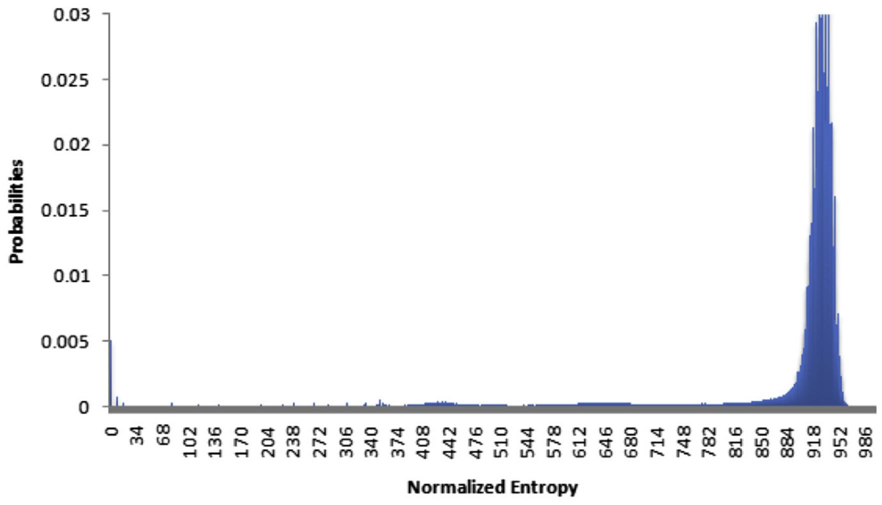


Fig. 11 – Entropy Distribution of 1500 Benign APK Files with $B = 128$.

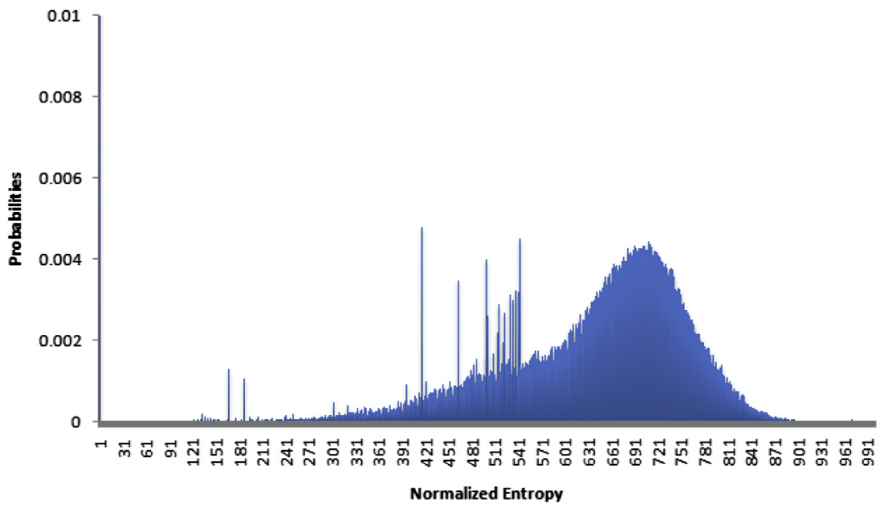


Fig. 12 – Entropy Distribution of 1500 Benign DEX Files with $B = 64$.

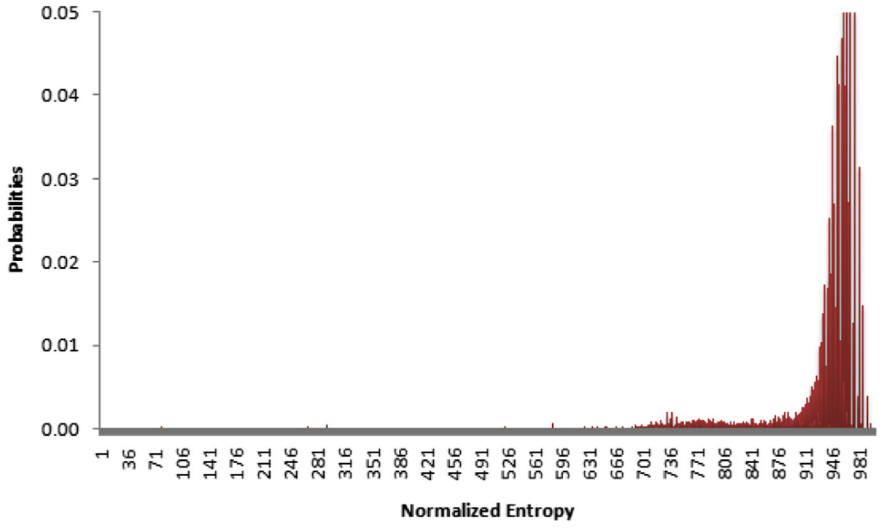


Fig. 13 – Entropy Distribution of 1242 Malicious APK Files with $B = 64$.

only DEX file can generate statistically robust features to differentiate malicious apps from benign one, efficiency of scanning and detection shall improve. Fuzzy-hashing based approach (Zhou et al., 2012a) has been successfully applied on just Dalvik-opcodes within DEX files to detect repackaged apps. Following the insight, we also evaluated performance of AndroSimilar on just DEX files. R_{prec} table has been generated from entropy distributions of both benign and malicious DEX files as shown in Figs. 12 and 15 respectively. Normalized entropy distribution of DEX files is visibly more spread than its APK counterparts. There is negligible concentration at the higher entropy end; weak features are removed at the lower entropy end only.

5.2.1. *Detecting unknown variants*

We extracted *classes.dex* files from our dataset of 3309 malicious apps. We divided those DEX files into two sets, first-one with 2854 DEX files as a signature-set and second-one with 455 DEX files as unknown variants of malicious ones. Result is shown in Table 9, where first row represents unknown variants with sixth column showing the detection rate for $B = 64$. TP rate is 72.25%, which is nearly 4% less than their APKs counterpart in fourth column.

5.2.2. *Detecting obfuscated malware*

We extracted *classes.dex* files out of obfuscated samples we generated previously. Table 9, in which rows including and after the third-one shows the detection rate for each obfuscation types separately for $B = 64$ in the sixth column. TP rate is merely 5.23%, which is nearly 75% less than its APKs counterpart in fourth column. Thus, AndroSimilar performed worst for only considering obfuscated DEX files, contrary to fuzzy-hashing based approach used by Zhou et al. (2012a).

5.3. *Signature-set reduction*

We evaluated our custom signature-set reduction algorithm on 49 malware families of total 1242 samples taken from Malware Genome Project (Yajin Zhou, 2013). Caveat of applying this algorithm is that we must have all malware samples categorized into corresponding families. We have been successfully able to reduce signatures on an average of nearly 42% per family basis, as shown in Fig. 8. Highest reduction is 88% for *Gone60* family, next is 87% for *AnserverBot* family. That means, we can detect all the 1242 malicious samples using just 519 signatures, which can save us a lots of space. Reducing the number of signatures in this way may affect the detection rate of new variants. For example, in this case, AndroSimilar will be able to detect only those variants, which are similar to those 519 signature-set only.

5.4. *Comparison with SSDEEP*

SSDEEP (Kornblum, 2006) is a byte-level Context Triggered Piece-wise Hashing (CTPH) approach that finds similarity

Table 1 – An example: similarities between a set of apps.

SDHash similarity score(%)	App name	App name
70	A_1	A_2
65	A_1	A_3
90	A_2	A_3
94	A_4	A_5
97	A_5	A_6
20	A_2	A_5
30	A_1	A_6

Table 2 – Distance-matrix in initial state.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
A_1	0	30	35	100	100	70	100	100
A_2	30	0	10	100	80	100	100	100
A_3	35	10	0	100	100	100	100	100
A_4	100	100	100	0	5	100	100	100
A_5	100	80	100	5	0	3	100	100
A_6	70	100	100	100	3	0	100	100
A_7	100	100	100	100	100	100	0	100
A_8	100	100	100	100	100	100	100	0

Table 3 – Distance-matrix after one iteration.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	N_c	T_{cost}
A_1	0	30	35	100	100	70	100	100	2	65
A_2	30	0	10	100	80	100	100	100	2	40
A_3	35	10	0	100	100	100	100	100	2	45
A_4	100	100	100	0	5	100	100	100	1	5
A_5	100	80	100	5	0	3	100	100	2	8
A_6	70	100	100	100	3	0	100	100	1	3
A_7	100	100	100	100	100	100	0	100	0	–
A_8	100	100	100	100	100	100	100	0	0	–

Table 4 – Distance-matrix after forming cluster A5.

	A_1	A_2	A_3	A_7	A_8	N_c	T_{cost}
A_1	0	30	35	100	100	2	65
A_2	30	0	10	100	100	2	40
A_3	35	10	0	100	100	2	45
A_7	100	100	100	0	100	0	–
A_8	100	100	100	100	0	0	–

between files, whereas, AndroSimilar finds robust byte-sequences as features based on SDHash to capture content homogeneity among similar files. However, SSDEEP generates a fixed length of 80 bytes as a signature for any file size, in contrast, AndroSimilar generates variable length signature. Vassil Roussev (Roussev, 2011) has done comprehensive evaluation of the performance between SSDEEP and SDHash. Due to fixed length signature, SSDEEP's performance is strictly dependent on the file size. For example, consider a file of size 1 MB, SSDEEP can reliably correlate that file embedded in a large file not greater than 3 MB, whereas, SDHash has the upper limit of 1 GB for the large file. Conclusively, Vassil Roussev found that SDHash outperforms SSDEEP with precision rates of 94% and 68% respectively (Roussev, 2011).

6. Discussion

Evaluation of results demonstrate AndroSimilar's effectiveness in detecting zero-day variants as well as code-obfuscated samples of known malware families. Moreover, signature generation is completely automated, which scales effectively given the rise of malicious Android apps. We checked samples collected from Google Play and third-party app-stores against signature set of 3309 malware for both $B = 64$ and $B = 128$, result is shown in Fig. 10. 157 samples from Google Play and 152 samples from third-party app-stores are detected as malicious in-case of $B = 64$. By manual analysis, we found that those samples are completely benign. Furthermore, 43 apps in Google Play and 128 apps in third-party app-stores are the original versions of their malicious counterparts, means, malware authors reverse-engineered the original apps from app-stores, added malicious payloads into it and repackaged them using the tools mentioned in Section 2.2. Rest of the detected samples were indeed false-positives, which amounts to 0.71% in Google Play and 0.47% in third-party app-stores. Fig. 11 shows, what changes have been done by malware authors in six benign apps from third-party app-stores through reverse-engineering and repackaging. These facts indicate the in-effectiveness of AndroSimilar in confirming whether the detected app is itself the original one, from which malicious one was repackaged. Ironically, side effect of this result is that this approach can successfully be applied by app-stores to detect repackaged apps.

Signature size of a single app is also an important factor because of less storage space available on Smartphones. Signature size depends on R_{pop} threshold, which is used to retain rare but important features. More the threshold, less the number of features retained (Roussev, 2009), which ultimately reduces the overall size of signature. Similarly, byte-sequence length B also controls the size of signature. More the value of B, lesser the granularity (Roussev, 2009), which ultimately reduces the overall size of signature. We followed the suggestion of Roussev (2009) to use R_{pop} threshold as 16, and our results also showed that $B = 64$ gives better detection rate with reasonable performance.

Evaluation of AndroSimilar on just DEX files shows that it performs worst in-case of obfuscation with TP rate less than 4%, meanwhile it is able to detect successfully the corresponding obfuscated malicious APKs as a whole with TP rate of more than 80%. This happens because obfuscation changes just DEX file, whereas other parts of an app such as resources, assets and manifest are completely untouched. Nevertheless, syntactic layout of app still changes as a whole, but there remains common features to be exploited between the original and obfuscated app. We have evaluated results for each obfuscation technique separately, which is rather a simplified assumption. Because all obfuscation techniques together can also be applied to generate a malicious variant, which makes them more difficult to be detected.

We have kept the value of *inter-app* similarity threshold as 25 during experimentation because it gives reasonably less false-positives (Roussev). This similarity threshold should be decided according to the requirement, which has an impact on True Positives (TP), correct detection of malicious apps, as well as False Positives (FP), incorrect detection of benign apps. As shown in Fig. 16, as similarity threshold increases, FP decreases, but at the cost of TP.

Table 5 – Distance-matrix after forming cluster A2.

	A_7	A_8	N_c	T_{cost}
A_7	0	100	0	–
A_8	100	0	0	–

7. Related work

Android malware outbreak has reached from 3 families (50 samples) in 2010 to nearly more than hundred families consisting thousands of sophisticated samples capable of causing considerable harm to the security of Android OS. Software similarity is a measurement to detect plagiarism at file level to detect similar content. Our approach works at file level to search strong statistical features that can differentiate between a malware and normal Android app. Features extracted from known malware families are useful in detecting

Table 6 – 15993 Google Play apps.

Category	#Apps	Category	#Apps	Category	#Apps	Category	#Apps
Arcade & action	638	Education	600	Music & audio	535	Sports	491
Books & reference	593	Entertainment	485	News & magazines	552	Sports games	415
Brain & puzzle	642	Finance	579	Personalization	642	Tools	731
Business	524	Health & fitness	614	Photography	498	Transportation	519
Cards & casino	541	Libraries & demo	435	Productivity	661	Travel & local	606
Casual	642	Lifestyle	528	Racing	233	Weather	405
Comics	325	Media & video	596	Shopping	458		
Communication	619	Medical	356	Social	530		

Table 7 – 5139 Third-party apps.

App Store	#App	App Store	#Apps
android.d.cn (Android.d.cn, 2014)	489	gfan.com (gfan.com, 2014)	1733
hiapk.com (Hiapk.com, 2014)	534	mumayi.com (Mumayi.com, 2014)	2363

Table 8 – 3309 Malicious App dataset. It also shows the result of our Signature-set Reduction Algorithm with average reduction of 42% signatures per family.

Malware family	#Apps	Reduced No. of signatures	Reduction rate (%)	Malware family	#Apps	Reduced no. of signatures	Reduction rate (%)
GingerMaster	**4**	2	50.00	Geinimi	**69**	63	8.70
ADRD	**22**	14	36.36	GoldDream	**47**	42	10.64
AnserverBot	**187**	23	87.70	Gone60	**9**	1	88.89
Asroot	**8**	6	25.00	GPSSMSSpy	**6**	5	16.67
BaseBridge	**122**	22	81.97	HippoSMS	**4**	3	25.00
BeanBot	**8**	4	50.00	jSMSHider	**16**	12	25.00
Bgserv	**9**	3	66.67	KMin	**52**	10	80.77
CruseWin	**2**	2	0.00	NickySpy	**2**	2	0.00
DroidDream	**16**	14	12.50	Pjapps	**58**	35	39.66
DroidDreamLight	**46**	44	4.35	Plankton	**11**	9	18.18
DroidKungFu1	**34**	11	67.65	RogueLemon	**2**	2	0.00
DroidKungFu2	**30**	14	53.33	RogueSPPush	**9**	2	77.78
DroidKungFu3	**309**	98	68.28	SndApps	**10**	5	50.00
DroidKungFu4	**96**	45	53.13	YZHC	**22**	5	77.27
DroidKungFuSapp	**3**	1	66.67	zHash	**11**	6	45.45
FakePlayer	**6**	4	33.33	Zsone	**12**	10	16.67
Others	**2067**	NA	NA				

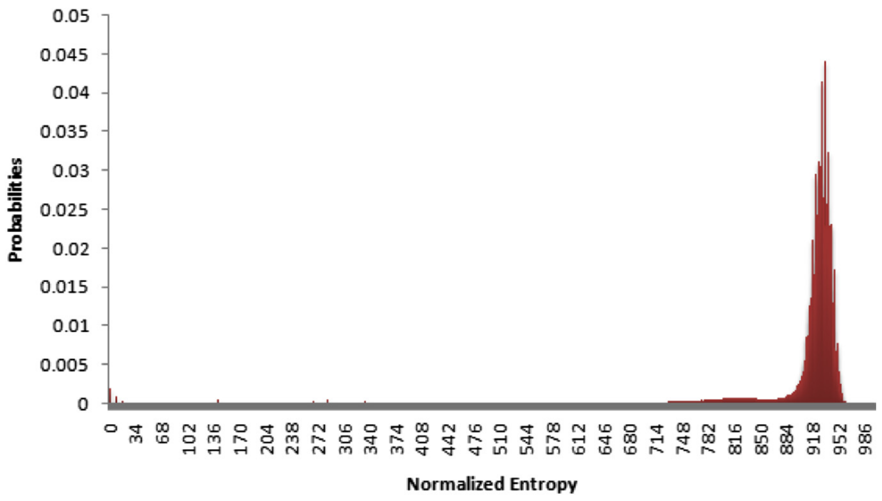


Fig. 14 – Entropy Distribution of 1242 Malicious APK Files with $B = 128$.

Table 9 – Results for malware variants and obfuscated malware.

Category	Signature set	Test set	% TP APKs 64 byte features	% TP APKs 128 byte features	% TP DEX 64 byte features
Unknown Malware	**2854**	**455**	**76.48**	**73.85**	**72.25**
Code Obfuscation					
Method Renaming	3309	234	81.62	77.35	5.56
Junk Method Insertion	3309	234	80.77	76.92	5.13
GOTO Obfuscation	3309	230	82.17	78.70	6.52
String Encryption	3309	158	79.11	73.42	8.23
All Obfuscations at Once	3309	157	78.34	71.97	0.00
Overall	**3309**	**1013**	**80.65**	**76.11**	**5.23**

Table 10 – Detection results of Google Play and third-party apps.

Category	Signature set	Test set	% FP APKs 64 byte features	% FP APKs 128 byte features
Google Play	3309	15993	0.98	0.78
Third-Party	3309	5139	2.96	2.30

unknown variants of known malware families as well as obfuscated malware. AndroSimilar is different from DroidMOSS (Zhou et al., 2012a), which is based on fuzzy hashing technique known as Context Trigger Piecewise Hashing, generating a fixed 80 byte signature to detect repackaged apps. Our approach generates normalized entropy features such that they rarely occurs in two unrelated files, but occurs in related or similar files. Source code of DroidMOSS is not available, so we could not compare the performance against AndroSimilar. However, as we covered in Section 5.4, AndroSimilar may potentially perform better than DroidMOSS. DroidMOSS just considers `classes.dex` for the comparison, whereas, AndroSimilar considers whole APK. We observed that repackaged apps has almost the same resources of their corresponding original apps, but bytecode may get changed drastically using obfuscation and other modifications. We believe in that case AndroSimilar will outperform DroidMOSS.

Our signature generation technique is based on Vassil Roussev's SDHash algorithm for extracting statistically robust features for finding similarity among different documents (Roussev). Roussev applied this concept in digital forensics as a means to quickly, accurately and reliably correlate digital artifacts (Roussev, 2011). In Vijay et al. (2011) the authors have considered entropy features of byte blocks starting from 500 upto 10,000 incrementally to determine packed Windows binary. Their approach is interesting but depends on how the software packer inserts packer-code in the executable. Considering insertion just at the beginning or ending may fail as packer can insert its code randomly anywhere.

DroidAnalytics (Zheng et al.) proposed a static analysis framework for effective detection of obfuscated Android malware. Signature generation is done at three levels namely method, class and app level in that sequence to detect obfuscated or repackaged malware. The approach is effective against simple obfuscation techniques like method renaming, control flow goto-obfuscation and string encryption. This framework for static analysis does not discuss the time taken for analyzing apps.

Permissions in Android are broadly categorized into normal, dangerous and signature/system permissions, among them dangerous permissions control access to sensitive APIs within Android application framework. Barrera et al. (2010) proposed a mechanism to detect malicious code based on distribution of permissions requested by apps of varied categories. Felt et al. (2011) also investigated issues in permission escalations to detect the malicious content from official as well as reliable third party markets. Woodpecker tool (Grace et al., 2012a) analyzed the apps to identify the dangerous permissions that may compromise the OS.

Xiang et al. (Yajin and Xuxian, 2012) performed an extensive study of their malware repository by characterizing them according to the functionality. They were able to identify 211 real malware. DroidRanger approach applied footprint mechanism at the opcode level to find out malicious family logic. Although the approach is interesting it has an overhead of disassembling and detail opcode analysis is not suitable for real-time scanning. Similarly, they also developed (Zhou et al., 2012a) fuzzy hashing approach to detect repackaged apps considering only Dalvik-opcodes within DEX files. The approach seems time consuming for real-time detection as there is a disassembly overhead. Moreover, fuzzy hashing generates a fixed 80 bytes signature and is effective only for smaller file-size. Our approach generates variable-size signature, which scales for files of any size.

RiskRanker (Grace et al., 2012b) analyzes the behavior of an app to identify if it has dangerous logic to compromise the security. The approach relies on class path to identify any mutation, but remains ineffective against obfuscation techniques.

TaintDroid (William et al., 2011) is a dynamic analysis tool used that monitors executing apps to find out privacy leakage caused by an app. In Enck et al. (2009), authors developed a rule-set to stop risky apps from installation, based on combination of dangerous permissions requested by them.

DroidRanger (Zhou et al., 2012b) applies combination of static and dynamic analysis for malicious code detection.

Table 11 – Analysis of Third Party apps detected as malicious by AndroSimilar.

Third Party App	Malicious App	Comment
SHA-1: 2ea5f6dc465cf89caab438ae8bcb5b42de3e444f MD-5: ee8be1b7f2761f42957bb22b2a0f6414 Package Name: com.requiem.armoredStrike	SHA-1: bdd581d2d57ad71b0bf6401e76c8120cd5dc0173 MD5: 5d27c7d0c5630f4c7a8b7a8f45512f09 Package Name: com.requiem.armoredStrike **Malware Family: Geinimi**	**Disguised as updated version** Original Main launcher receiver changed **Dangerous permissions added:** ACCESS_FINE_LOCATION, ACCESS_GPS, CALL_PHONE, READ/WRITE CONTACTS READ/SEND_SMS, READ/WRITE HISTORY_BOOKMARKS.
SHA-1: fc438c6d0cabc2190c26b41e9dc5d3d3843274eb MD5: ec3b45dc3ebda87cc420722f1895e75c Package Name: com.camelgames.hyperjump	SHA-1: 00983aad12700be0a440296c6173b18a829e9369 MD5: 513971a8cde07e145a85a8707f83e4b5 Package Name: com.camelgames.hyperjump **Malware Family: Pjapps**	**Service added:**Intent receivers for SMS_RECEIVED, WAP_PUSH_RECEIVED. **Dangerous permissions added:** RECEIVE_SMS, RECEIVE_MMS, SEND_SMS, NEW_OUTGOING_CALL, READ/WRITE_HISTORY_BOOKMARKS, INSTALL_PACKAGES.
SHA-1: 97bc745f247da451995a0be635150eb71a3c0f2f MD5: 07e640792506d889b67e4a7061a9aff7 Package Name: jp.hudson.android.militarymadness	SHA-1: 1f522d9ab07fc716e7f201fad12ccea396987a83 MD5: 6ea15fdda8ba208b19e5c7131e9d413f Package Name: jp.hudson.android.militarymadnes **Malware Family: GoldDream**	**Very minor change in package name and activities added.** Intent receiver for BOOT_COMPLETED, SMS_RECEIVE, PHONE_STATE, NEW_OUTGOING_CALL added. **Dangerous Permissions added:** RECEIVE_SMS, READ_SMS, INSTALL_PACKAGES, RECEIVE_BOOT_COMPLETED, HISTORY_BOOKMARKS.
SHA-1: 2b11e7d3bc8421da143deab57acaea03e0a83b1c MD5: 21d81b064951e9ed7beff98d6879e55a Package Name: jpcom.wuxi.GoldMiner.domob	SHA-1: b9d992b88ef1a4a75362f8f5d069716ea7a3321e MD5: 025a55c1bcbd3be2ca03aa314ce9a4c2 Package Name: jpcom.handcn.GoldMiner.free **Family: Geinimi**	**Original Main launcher receiver changed.** Advertisement Publisher ID changed, along with the ad-library **Dangerous permissions added:** CALL_PHONE, INSTALL_SHORTCUT, READ/WRITE_CONTACTS, SEND/READ_SMS, READ/WRITE_HISTORY_BOOKMARKS, ACCESS_GPS, ACCESS_LOCATION.
SHA-1: 9ef6fc25d9e599ce6bfa7c3fb4d21a775f5cf98f MD5: c9caebc6cb727d720e04cb77ce1e042e Package Name: com.camelgames.mxmotorlite	SHA-1: ef140ab1ad04bd9e52c8c5f2fb6440f3a9ebe8ea MD5: e5b7b76bd7154dea167f108daa0488fc Package Name: com.camelgames.mxmotor **Family: AnserverBot**	**Many new activities and services added.** Receiver for SMS_RECEIVED, BOOT_COMPLETED, PICK_WIFI_WORK added. **Dangerous Permissions added:**READ/RECEIVE/WRITE_SMS, RECEIVE_BOOT_COMPLETED, READ/WRITE_CONTACTS, CALL_PHONE, READ_PHONE_STATE.
SHA-1: 218af28eeb168dd16df6d0faacb3f77f51fc66df MD5: 4b0c93b1a14b5fdad60715a8a6b1987e Package Name: com.moregame.drakula	SHA-1: b9891ab782cf643c81f0f1c130ea119384dbefe1 MD5: 8498984d8f9b7260fd032d6f0a2534aa Package Name: com.moregame.drakula **Family: Geinimi**	**Original Main launcher receiver changed.** **Receiver for BOOT_COMPLETED event added.** **Dangerous permissions added:** ACCESS_FINE_LOCATION, CALL_PHONE, READ/WRITE_CONTACTS, READ/SEND_SMS, READ/WRITE_HISTORY_BOOKMARKS, ACCESS_GPS, ACCESS_LOCATION added.

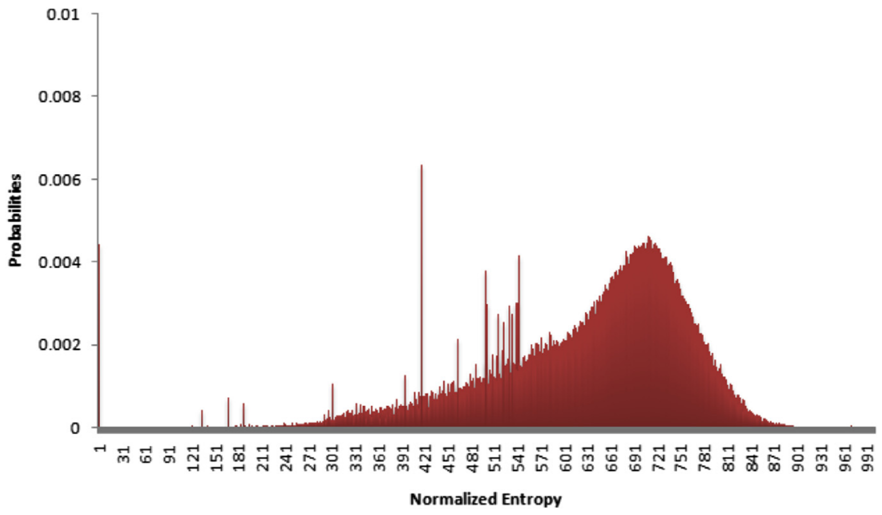


Fig. 15 – Entropy Distribution of 1242 Malicious DEX Files with $B = 64$.

Complementary approach helps in observing certain features that may not be possible with a single approach.

8. Conclusion

In this paper, we aim to detect variants of known Android malware families, obfuscated malware and repackaged apps from app-stores by presenting AndroSimilar prototype. Proposed method utilizes statistically robust features generated using SDHash mechanism to automatically create variable-length signatures from apps. AndroSimilar effectively detects existing and variants of known malware families with accuracy of more than 76%. It is also able to detect reasonably, existing malware, obfuscated with techniques like string encryption, method renaming, junk method insertion and changing control flow, which helps them evade cryptographic AV-signatures, with accuracy of more than 80%. We also found that some of the false-positives generated from Google Play and third-party app-stores were actually the original apps of their malicious counterparts. This suggests that AndroSimilar can be applied to detect repackaged apps from app-stores. Our evaluation is also consonant with the suggestion given by Vassil Roussev (Roussev, 2009) that feature-size of 64-bytes is more suitable considering the performance against feature-size of 128-bytes. We evaluated our custom algorithm to reduce signature-set against 49 malware families from Malware Genome Project (Yajin Zhou, 2013), in which, we are able to reduce upto more than 40% signatures. We also applied AndroSimilar to just DEX files, in which, it performed worst in-case of obfuscated malware, thus proving our analysis considering whole APKs as appropriate.

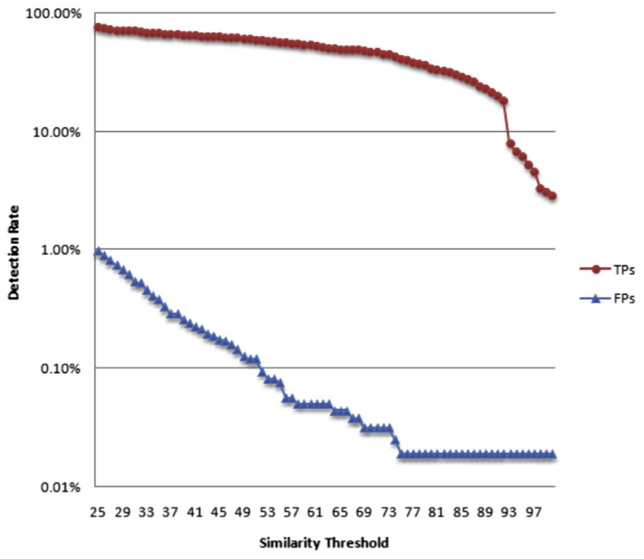


Fig. 16 – Effect of changing similarity threshold on True-Positives and False-Positives.

REFERENCES

A. Inc. Class to Dex Conversion with Dx. 2013. http://developer.android.com/tools/help/index.html [Online accessed 05.03.13].

Amazon Inc. Amazon appstore for android. 2013. http://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011 [Online accessed 27.06.13].

Andre G, Ramos P. BOXER SMS trojan. Tech. rep. ESET Latin American Lab; 2013.

Android.d.cn. Third-party app-store, China. 2014. http://www.android.d.cn/ [Online accessed 21.01.14].

APKTool. Reverse engineering with apktool. 2013. https://code.google.com/android/apk-tool [Online accessed 20.03.13].

BakSmali. Reverse engineering with smali/baksmali. 2013. https://code.google.com/smali [Online accessed 20.03.13].

Barrera D, Kayacik HG, van Oorschot PC, Somayaji A. A methodology for empirical analysis of permission-based security models and its application to android. In: Proceedings of the 17th ACM conference on Computer and communications security, CCS '10. New York, NY, USA: ACM; 2010. p. 73–84.

Castillo CA. Android malware past, present, and future. Tech. rep. MSWC; 2012.

Dex2Jar. Adroid decompiling with Dex2jar. 2013. http://code.google.com/p/dex2jar/ [Online accessed 15.05.13].

Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. ACM; 2009. p. 235–45.

Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11. New York, NY, USA: ACM; 2011. p. 627–38.

G. Inc. Android smartphone sales forecast. 2013. http://www.gartner.com/newsroom/id/2645115 [Online accessed 17.06.13].

G. Play. Official android market. Google Play; 2013. https://market.android.com/ [Online accessed 17.06.13].

gfan.com. Third-party app-store, China. 2014. http://www.gfan.com/ [Online accessed 21.01.14].

Grace M, Zhou Y, Wang Z, Jiang X. Systematic detection of capability leaks in stock android smartphones. In: Proceedings of the 19th Network and Distributed System Security Symposium (NDSS); 2012.

Grace M, Zhou Y, Zhang Q, Zou S, Jiang X. RiskRanker: scalable accurate zero-day android malware detection. In: 10th International Conference on Mobile Systems, Applications, and Services, MobiSys. NY, USA: ACM; 2012b. p. 281–94.

Hiapk.com. Third-party app-store, China. 2014. http://www.hiapk.com/ [Online accessed 21.01.14].

JAD. Java decompiler. 2013. http://www.varaneckas.com/jad [Online accessed 20.03.13].

Kornblum J. Identifying identical files using context triggered piecewise hashing. Digital Investigation 2006;3:91–7.

L. Inc.. State of mobile security 2012. Tech. rep. Lookout Mobile Security; 2012.

Mumayi.com. Third-party app-store, China. 2014. http://www.mumayi.com/ [Online accessed 21.01.14].

Oberhide J. Dissecting the android bouncer. 2012. http://jon.oberheide.org/blog/2012/06/21/ [Online accessed 01.06.12.

ProGuard. Java bytecode obfuscator. 2013. http://proguard.sourceforge.net/ [Online accessed 20.03.13].

V. Roussev, Data fingerprinting with similarity hashes, Advances in digital forensics.

Roussev V. Building a better similarity trap with statistically improbable features. In: System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on, IEEE; 2009. p. 1–10.

Roussev V. An evaluation of forensic similarity hashes. Digit Investig 2011;8:S34–41.

Vijay L, Manoj Singh G, Parvez F, Smita N. PEAL-packed executable analysis. In: Proceedings of International Conference on Advanced Computing Networking and Security, ADCONS '11. SPRINGER; 2011.

William E, Peter G, Byunggon C, Landon C. TaintDroid : an information flow tracking system for realtime privacy monitoring on smartphones. In: USENIX Symposium on Operating Systems Design and Implementation. USENIX; 2011.

Yajin Z, Xuxian J. Dissecting android malware: characterization and evolution. In: Proceedings of the 33rd IEEE Symposium on Security and Privacy 2012. Oakland: IEEE; 2012.

Yajin Zhou XJ. Malware genome project. 2013. http://www.malgenomeproject.org/ [Online accessed 15.05.13].

Zheng M, Lee PPC, Lui JCS. ADAM: an automatic and extensible platform to stress test android anti-virus systems. In: DIMVA; 2012. p. 82–101.

Zheng M, Sun M, Lui J. C. S., DroidAnalytics: a signature based analytic system to collect, extract, analyze and associate android malware, CoRR.

Zhou W, Zhou Y, Jiang X, Ning P. Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of Second ACM conference on Data Application Security and Privacy, CODASPY '12, New York, USA; 2012. p. 317–26.

Zhou W, Zhou Y, Jiang X. Hey, you get off my market: detecting malicious apps in official and third party android markets. In: Annual Network and Distributed Security Symposium, NDSS, New York, NY, USA; 2012.