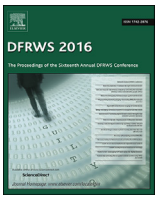
Digital Investigation 18 (2016) S33–S45

Contents lists available at ScienceDirect

Digital Investigation

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



DFRWS USA 2016 — Proceedings of the 16th Annual USA Digital Forensics Research Conference

Fingerprinting Android packaging: Generating DNAs for malware detection

ElMouatez Billah Karbab*, Mourad Debbabi, Djedjiga Mouheb

Computer Security Laboratory, Concordia University & NCFTA-Canada, Montreal, Quebec, Canada

Keywords:
Fingerprinting
Malware
Mobile
Android
Fuzzy hashing
Detection
Family attribution

A B S T R A C T

Android's market experienced exponential popularity during the last few years. This blazing growth has, unfortunately, opened the door to thousands of malicious applications targeting Android devices everyday. Moreover, with the increasing sophistication of today's malware, the use of traditional hashing techniques for Android malware fingerprinting becomes defenseless against polymorphic malicious applications. Inspired by fuzzy hashing techniques, we propose, in this paper, a novel and comprehensive fingerprinting approach for Android packaging *APK*. The proposed fingerprint captures, not only the binary features of the *APK* file, but also the underlying structure of the app. Furthermore, we leverage this fingerprinting technique to build ROAR, an automatic system for Android malware detection and family attribution. Our experiments show that the proposed fingerprint and the ROAR system achieve a precision of 95%.

Introduction

In recent years, we have witnessed a phenomenal popularity and growth of Android devices. It is estimated that 2 billion devices are currently powered by Android OS (Ericsson, 2013). This trend is expected to continue to reach more than 5.6 billion devices by 2019 (Ericsson, 2013). Due to their proliferation and ubiquitousness, Android devices have become a tempting target for cyber criminals. According to a *Cisco* report (Cisco, 2014), mobile malware mostly targets Android devices. In this setting, the need to develop effective and accurate forensics methods, techniques and tools for the detection and analysis of Android malware becomes a desideratum.

To address the malware variation flood, multiple defence mechanisms have been proposed by the *anti-mobile-malware* industry, with *signature-based detection* being the most adopted technique. The latter uses malware digest or *signature* to match against mobile applications in order to detect any malicious code. Traditional cryptographic hashing algorithms such as *SHA1* and *MD5* have been widely adopted for generating malware signatures. Cryptographic hashing methods have the advantage of being simple and fast. They are, however, highly sensitive to even small changes, which makes these methods defenseless against malware variations. Moreover, despite its effectiveness, signature-based detection could be easily defeated by new malicious applications with only tiny modifications as is the case with *polymorphic* attacks.

To overcome the drawbacks of cryptographic hashing, a new technique, namely *fuzzy hashing*, has emerged in the literature. The concept of fuzzy hashing was first introduced as *ssdeep* (Kornblum, 2006) in *rsync checksum* (Tridgell and Mackerras). The main advantage of this technique over cryptographic hashing lies in its tolerance to changes. Thanks to this important property, fuzzy hashing has been widely leveraged to detect Web-based document duplication (Figuerola et al., 2011). In cyber security, fuzzy hashing has been mainly embraced in malware fingerprinting. For instance, Virus Total has been using the *ssdeep* fuzzy hashing for malware fingerprinting since 2012. There

* Corresponding author.
E-mail address: e_karbab@encs.concordia.ca (E.B. Karbab).

http://dx.doi.org/10.1016/j.diin.2016.04.013

are several other attempts, such as *mvHash-B* (Breitinger et al., 2013), *dcfldd* (DCFL, 2016), *sdhash* (sdhash 2016; Roussev), and *mrshv2* (Breitinger and Baier, 2013), to apply fuzzy hashing in multiple applications. In the context of malware detection, it consists of two steps: i) malware binary digest computation, ii) matching the digest against other malware samples.

Despite its effectiveness compared to cryptographic hashing, fuzzy hashing technique suffers from some limitations. First, it ignores the underneath structure and semantics of the malicious package. Second, fuzzy hashing suffers from its single fingerprint bounded with a maximum size (e.g., *ssdeep* (Roussev, 2010)), thus preventing packages with different sizes and features to be effectively compared. Other drawbacks of fuzzy hashing related to specific algorithms are presented in (Li et al., 2015) concerning *mvHash-B* fuzzy hash algorithm (Breitinger et al., 2013). Furthermore, the compressed nature of Android *APK* package makes the repacking of malicious apps an easy task. Moreover, the increasing number of Android app stores and the lack of security verification in some stores increase the chance of attackers to deploy malicious applications in multiple stores. Another issue is related to *native libraries* (Wang and Shieh, 2015), specifically at the level of *Java objects*, executed on top of *Dalvik* machine. It has been shown that these libraries are exploited by a significant number of Android malware, such as the well-known sophisticated *DroidKungFu* malware and its many variations (Zhou and Jiang, 2012). Not analyzing the native library would make the distinction between its variations a very challenging task.

Our objectives are to: i) Develop a more accurate, yet broad, fuzzy fingerprinting technique for Android OS malware. The proposed fingerprint relies on a customized fuzzy hashing technique that addresses the previous limitations. ii) Design and implement a framework for Android malware detection and family attribution on top of the developed fuzzy fingerprint. To this end, we propose APK-DNA, a fuzzy fingerprint that captures both the structure and the semantics of the *APK* file using most Android *APK* features. This fingerprint covers: i) the underneath Android app structure, including both *Dalvik machine byte-code* ii) the meta-data of the Android app. Our empirical results indicate that our fingerprinting approach is highly robust to app changes and accurate in terms of fingerprint computation and matching.

Moreover, we build ROAR, an automatic framework for Android malware detection, using the proposed APK-DNA. The goal is to generate fingerprints for known Android malicious apps, and then detect new malware variations using similarity computing. In ROAR framework, we propose two different approaches for malware detection: i) *family-fingerprinting*, and ii) *peer-matching*. In addition to malware detection, we aim to attribute the family lineage of the mobile malware. To this end, ROAR attributes a similarity score to each possible variation in the same malware family. We evaluate APK-DNA and ROAR using real malware samples from the Android Malware Genome Project (Android Malware Genome Project, 2015; Zhou and Jiang, 2012). We experiment with multiple Android malware families. Our evaluation demonstrates that ROAR is highly accurate compared to state of-the-art approaches.

This paper makes the following contributions:

- We propose a novel and rather comprehensive fingerprinting technique for Android application packages (APK) based on fuzzy hashing. The proposed fingerprint considers not only the binary format of *APK* but also its structure and semantics.
- We design and implement ROAR, a framework that leverages the proposed fingerprinting technique for Android malware detection, following two different approaches, namely *peer-matching* and *family fingerprint*. In addition, ROAR is able to detect malware variations and attribute the family of the detected malware.
- We evaluate ROAR on 928 Android malware samples from (Android Malware Genome Project, 2015; Zhou and Jiang, 2012) dataset. The evaluation results demonstrate the high accuracy of ROAR in terms of both malware detection and family attribution.

The remainder of this paper is organized as follows: Section Approach overview presents an overview of the proposed approach. Section APK-DNA fingerprint is dedicated to the APK-DNA fingerprinting. Section ROAR framework presents the ROAR framework. Section Experimental results details our experimental results. Section Limitations and future work discusses the limitations of the proposed approach together with some ideas on future research. The related work is reported in Section Related work. Section Conclusion contains some concluding remarks.

Approach Overview

Current fuzzy fingerprints such as *ssdeep* are computed against the app binary as a whole, which makes them ineffective for detecting malicious app variations. This problem gets even worst in case of Android OS because of the structure of apps packaging, which contains not only the actual compiled code but also other files such as media ones. To overcome this limitation, we propose an effective and broad fuzzy fingerprint that captures, not only binary features, but also the underneath structure and semantics of the *APK* package.

Accordingly, our approach for computing Android app fingerprints relies on decomposing the actual *APK* file into different content categories. For each category, we compute a customized fuzzy hash (sub-fingerprint). Note that for some categories, for instance *Dex* file, the application of the customized fuzzy hashing on the whole category content does not capture the structure of the underlying category. In this case, we apply fuzzy hashing against a selected *N-grams* of the category content. In our context, we use *byte n-grams* on binary files and *instruction n-grams* on assembly files. Furthermore, a best practice for malware fingerprinting is to increase the entropy of the app package content (Masud et al., 2007). To this end, we compress each category content before computing the

customized fuzzy hash on its N-grams. The resulting fuzzy hashes (sub-fingerprints) are then concatenated to produce the final composed fuzzy fingerprint, called APK-DNA. As depicted in Fig. 1, there are two main processes. First, we build a database of fingerprints by generating the APK-DNA for known malware samples. To identify whether a new app is malicious or not, and to know its family in case it is malicious, first we compute its APK-DNA and match it against the existing fingerprints in the database. Moreover, we use the proposed APK-DNA fingerprint as a basis to design and implement ROAR, a novel framework for malware detection and family attribution. ROAR's first approach, namely *family-fingerprinting*, computes a fingerprint for each malware family. Afterwards, it uses these family fingerprints to make security decisions on new apps. In the second approach, *peer-matching*, ROAR uses the whole fingerprint database for detection and attribution.

APK-DNA fingerprint

In this section, we present our approach for generating Android apps fingerprints. Before presenting the details of the fingerprint generation, it is necessary to understand the structure of the Android *APK* package.

Android APK format

Android application package (*APK*) is the official Android packaging format that is used for apps distribution and installation. By analogy, *APK* files are similar to *EXE* installation files for Windows or *RPM*/*DEB* files for Linux. More precisely, *APK* comes as a *ZIP* archive file, which contains the different components of the Android App. *APK* file content is organized into directories (namely **lib**, **res**, and **assets**) and files (namely **AndroidManifest.xml** and **classes.dex**). The purpose of each item is as follows: i) **AndroidManifest.xml** carries the app meta-data, e.g., name, version, required permissions, and used libraries. ii) The **classes.dex** contains the compiled classes of the Java code. iii) The **lib** directory stores C/C++ native libraries (NDK, 2016). iv) Finally, the resources directory contains the non-source code files that are packaged into the *APK* file during compilation. It mostly holds media files such as video, image, and audio files.

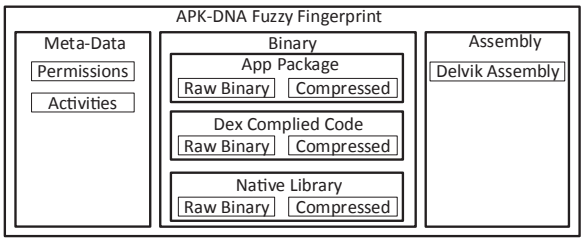


Fig. 2. Android package fingerprint structure.

APK-DNA fingerprint structure

We leverage the aforementioned *APK* structure to define the most important components for fingerprinting. The design of the *APK* fingerprint must consider most of its important components as unique features to be able to distinguish between different malware samples. As depicted in Fig. 2, APK-DNA is composed of three main sub-fingerprints based on their content type: **Metadata**, **Binary**, and **Assembly**. The **Metadata** sub-fingerprint contains information, which is extracted from the **AndroidManifest.xml** file. We particularly focus on the required permissions. The aim is to fingerprint the permission used for a specific Android malware or malware family. Our insight comes from the fact that some Android malware samples need a specific type of permission to conduct their malicious actions. For example, the malware **DroidKungFu1** (Zhou and Jiang, 2012) requires access to personal data to steal sensitive information. Having Android malware without access permissions to personal data, e.g., *phone number*, would suggest that this malware is most likely not part of **DroidKungFu1** family. Other metadata information could be considered for malware segregation, for instance, *Activity* list, *Service* list, and *Component*. In our current design of APK-DNA, we focus on the required access permissions.

The **Binary** sub-fingerprint captures the binary representation of the *APK* file content. In other words, we aim to fingerprint the byte sequence of Android malware. In this context, we use *n-grams* (Masud et al., 2007) as we will present in Section APK-DNA fingerprint generation. We divide the binary sub-fingerprint into three parts: **App Package**, **Dex Compiled Code**, and **Native Library**. The **App Package** consists of the *APK* file. Thus, all the components inside the package are considered (e.g., media file). Along with the raw *APK* package, we apply a compression schema

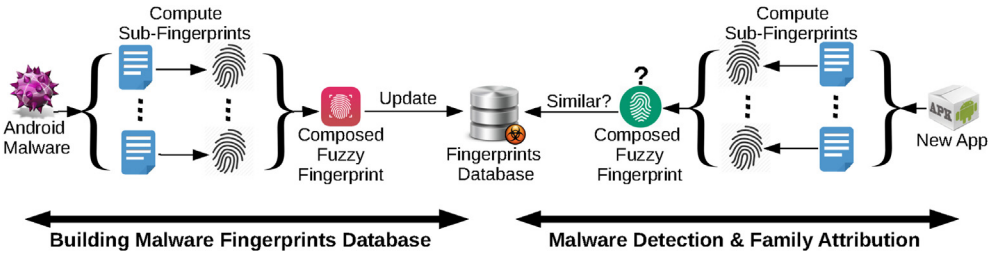


Fig. 1. Approach overview.

on the package to increase its *entropy* (Masud et al., 2007). In the **Dex Compiled Code**, we focus on the code section of the Android malware, which is located in the *classes.dex* file of Android apps. The use of the code section for malware detection has proven its accuracy (Jang and Brumley, 2011). In the context of Android malware, we use extracted features from the *classes.dex* as part of the APK-DNA. In addition, we use a high-entropy version of the *classes.dex* for fingerprinting by applying the compression. The **Native Library** part of the binary sub-fingerprint captures C/C++ shared libraries, used by malware. Using the native library for malware fingerprinting is essential in some cases, for example, to distinguish between two Android malware samples. For instance, if the malware uses a native library, it is more likely to be **DroidKungFu2** than **DroidKungFu1** because **DroidKungFu2** malware family uses C/C++ library and **DroidKungFu1** uses only *Java byte code*.

In the **Assembly** sub-fingerprint, we focus also on the code section of the Android malware, which is *classes.dex*. However, we do not consider the binary format. Instead, we use the reverse-engineered assembly code. As we will present in Section N-grams, we reverse engineered the *Dalvik byte-code* in order to extract instruction sequences used in the app. The **Assembly** sub-fingerprint aims to distinguish malware using the unique instruction sequences in the assembly file. We use the same technique as in the **Binary** sub-fingerprint, i.e., *n-grams*. However, here we consider the assembly instructions instead of bytes. In addition to assembly instructions, we could also consider *section names*, *call graph*, etc. In the current design, we focus on the assembly instructions for fingerprinting.

APK-DNA fingerprint generation

In this section, we present the computation steps for generating the APK-DNA fingerprint. In addition, we present the main techniques adopted in the design of the fingerprint, namely, *N-gram* and *Feature Hashing*.

N-grams

The *N-gram* technique is used for computing contiguous sequences of *N* items from a large sequence. For our purpose, we use N-grams to extract the sequences used by Android malware to be able to distinguish between different malware samples. To increase the fingerprint accuracy, we leverage two types of N-grams, namely *instruction N-grams* and *bytes N-grams*. As depicted in Fig. 4, the instruction N-grams are the unique sequences in the disassembly of *classes.dex* file, where the instructions are stripped from the parameters. In addition to instruction N-grams, we also use byte N-grams on different contents of the Android package. Fig. 4 illustrates different N-grams on both instructions and bytes of the first portion of the **AnserverBot** malware. We have experimented multiple options such as *Bigrams*, *3-g*, and *5-g*. The latter showed the best results in the design of APK-DNA fingerprint, as we will see in the evaluation section. The result of N-grams extraction is the list of unique *5-g* for each content category, i.e., *assembly instructions*, *classes.dex*, *native library*, and *APK file*.

Feature hashing

Feature hashing is a machine learning preprocessing technique for compacting an arbitrary number of features into a fixed-size feature vector. The feature hashing algorithm, described in Algorithm 1, takes as input the set of sequences generated by applying the N-grams technique and the target length of the feature vector. In the current implementation of APK-DNA, we use a bit feature vector of 16 KB but the size could be adjusted according to the needed density of the bit-feature vector to distinguish between apps. For example, the size of the assembly instruction vector could be less than the dex vector since the density produced by the instruction content is less than the dex one. Notice that in our implementation, we store only a binary value, which defines whether the N-gram exists or not. The standard feature hashing uses the frequency, i.e., the number of occurrences of a given N-gram. The output of the feature hashing algorithm is a feature-bit vector. Instead of using existing fuzzy hashing algorithms such as *ssdeep*, we leverage the feature vector as our fuzzy hashing technique for implementing APK-DNA fingerprint. In the next section, we present the complete process of computing the fingerprint by using N-grams and feature hashing as basic blocks.

Algorithm 1: Feature Vector Computation

```
input  : N-grams: Set,
         L: Feature Vector Length
output: Binary Feature Vector
features_vector = new bitvector[L];
for Item in N-grams do
    H = hash(Item) ;
    feature_index = H mod L ;
    features_vector[feature_index] = 1 ;
end
```

Fingerprint computation process

As shown in Fig. 3, the fingerprint computation process starts by decomposing the Android app *APK* file into four different content categories: 1) *Dalvik byte-code*, 2) *APK file*, 3) *native libraries*, and 4) *AndroidManifest file*. Each binary content is compressed to increase the entropy. Afterwards, we extract the bytes N-grams from the raw and the compressed content. The resulting N-grams set is provided as input to the feature hashing function to produce the customized fuzzy hashing. The size of each customized fuzzy hash is 16 KB as mentioned in Section Feature hashing. For *Dalvik byte-code*, we fingerprint the assembly code in addition to the binary fingerprint. First, we reverse engineer the *classes.dex* file to produce its assembly code. After prepossessing the assembly, we use the instruction sequence of the Android app to extract the instruction N-grams set. Afterwards, we use feature hashing to generate a 16 KB bit vector fingerprint for the assembly code. The current design of APK-DNA uses the *feature hashing* technique without *feature selection* because we aim to keep the maximum information on the targeted malware instance or its family. However, *feature selection* could be a promising technique to explore in future APK-DNA design.

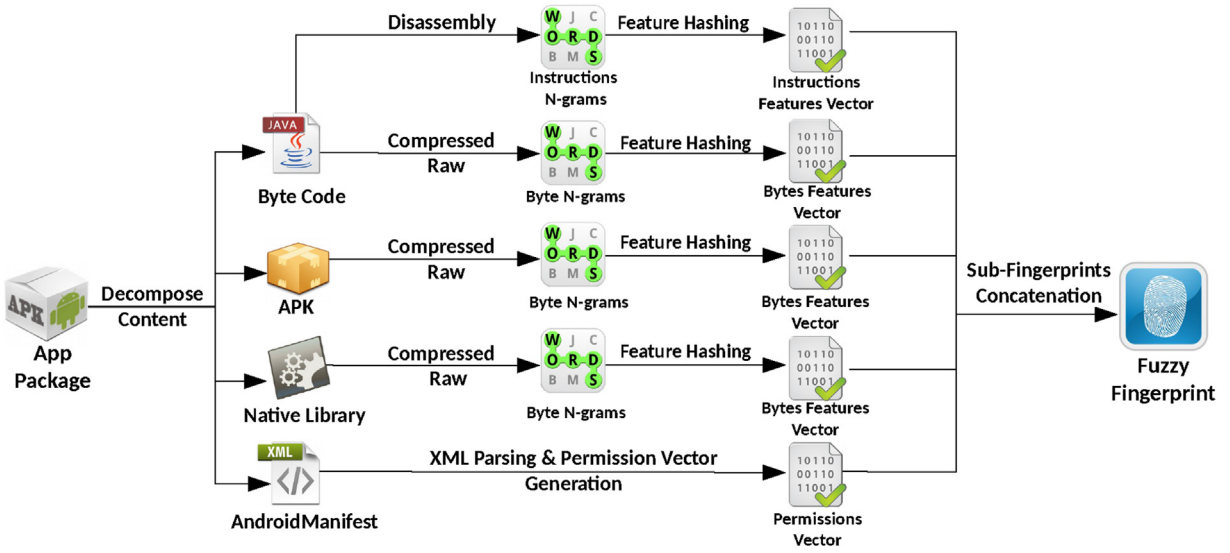


Fig. 3. APK-DNA computation process.

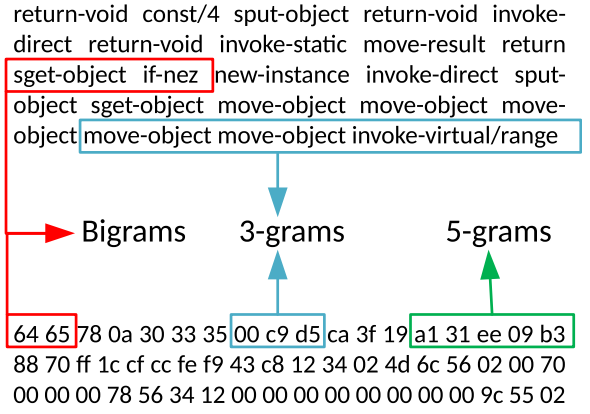


Fig. 4. First instructions and bytes of **AnserverBot** malware.

Regarding the *AndroidManifest file*, we first convert it into a readable format, then parse it to extract the required permissions by the Android app. To use the required permission app fingerprinting, we use a bit vector of all Android permissions in a predefined order. For a given required permission, we flag the bit to 1 in the permission vector if it exists in the *AndroidManifest file*. The result is a bit vector for all the permissions of the Android app. At the end of the aforementioned operations, we generate five bit vectors. The final step of the fuzzy fingerprint computation consists of concatenating all the produced digests into one fingerprint, designated as APK-DNA. It is important to mention that, for the purpose of similarity computation, we also keep track of the bits of each content vector. Notice that the content categories are mandatory for Android apps except the *native library*, which may not be part of the app. Therefore, we use a bit vector of zeros for the feature vector of the *native library*. The final size of APK-DNA is 16 KB for the feature vector of each content (there are 4 feature vectors: assembly, byte-code, APK, and native library). However, for the permission vectors, we use a 256-bit feature vector since Android permission system does not exceed this number.

Fingerprint similarity computation

The main reason of adopting the feature vector as a customized fuzzy hash is to make the similarity computation straightforward by using *Jaccard Similarity*, as shown in Equation (1). Since in our case, we have a set of bit feature vectors flagging the existence of a feature, we adopt *bitwise jaccard* similarity, as depicted in Equation (2).

$$Jaccard(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} \qquad 0 \le Jaccard(X,Y) \le 1 \tag{1}$$

$$Jaccard\_bitwise(A,B) = \frac{Ones(A \cdot B)}{Ones(A + B)} \qquad 0 \le Jaccard\_bitwise(X,Y) \le 1 \tag{2}$$

Let A and B be two bit-feature vectors, the union of the two vectors is given by the logical expression $A + B$ and its cardinality is the number of "1" bits in the resulting vector. Similarly, the cardinality of the intersection of the two vectors is the number of "1" bits in $A \cdot B$ bit vector. A complete example of similarity computation using bit-wise Jaccard similarity is shown in Fig. 5. As presented in Section Fingerprint computation process, APK-DNA fuzzy fingerprint is composed of five fuzzy hashes, which are bit-feature vectors. To compute the similarity between two

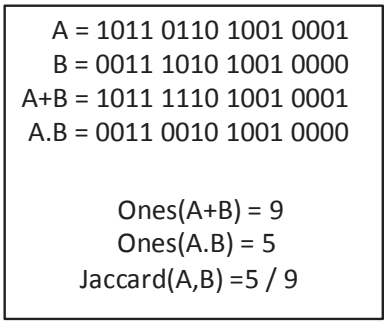


Fig. 5. Bitwise jaccard similarity computation.

fingerprints, we calculate the bit-wise Jaccard similarity between the bit feature vectors representing the same content. In other words, we calculate the similarity between the feature vectors of the assembly, byte-code, APK, native library, and permissions. The result is a set of five similarity values.

ROAR framework

In this section, we leverage the proposed APK-DNA fingerprint for Android malware detection. More precisely, we present i) *family-fingerprinting* approach, where we define and use a family fingerprint, and ii) *peers-matching* approach, where we compute the similarity between malware fingerprints. Both approaches are based on *peer-fingerprint-voting* mechanism to make a decision about malware detection and family attribution, as we present in the next sub-section.

Peer fingerprint voting technique

As we have seen in Section Fingerprint similarity computation, comparing two Android malware packages consists of computing similarities between their Metadata, Binary and Assembly sub-fingerprints, which gives numerical values on how the two packages are similar in a specific content category, as presented in Algorithm 2. In addition, we add the summation of all the similarities as a summary value of these sub-contents similarities. Note that other summary values such as the average and the maximum could also be used.

Algorithm 2: APK-DNA Similarity Computation

```
input  : APK-DNA A: list
         APK-DNA B: list
output: similarity-list: list
similarity-list = empty-list();
for content in content categories do
    similarity = Jaccard_bitwise(A[content],B[content]) ;
    similarity-list.add(similarity);
end
summation = sum(similarity-list);
similarity-list.add(summation);
```

Algorithm 3: Peer-Fingerprint Voting Mechanism

```
input  : similarity-list A-B: list
         similarity-list A-C: list
output: Decision
A-B-count = 0 ;
A-C-count = 0 ;
for content in content categories do
    if A-B[content] >A-C[content] then
        A-B-count += 1;
    else
        A-C-count += 1;
    end
end
if A-B-count >A-C-count then
    Decision = A-B;
else
    Decision = A-C;
end
```

However, it is unclear how to detect the most similar packages if we compare an unknown package to known malware packages using multiple sub-fingerprints. The most obvious solution is to merge the bit-vectors of each content category into one vector, and then compute the similarity of the resulting feature vector. However, in our case, merging the bit vectors will heavily reduce the contribution of some sub-fingerprints in the similarity computation. This is especially the case for the metadata sub-fingerprint, since the number of bits of the metadata feature vector is very small compared with the assembly ones. Similarly, the density of the assembly feature vector is considerably less than the binary one. Consequently, we propose to use a composed similarity using the *peer-fingerprint voting* technique. The idea is to compare between the parts (sub-fingerprints) instead of comparing the whole fingerprints, as depicted in Algorithm 3. In other words, we compare each sub-similarity pairs. The decision is made by a voting mechanism on the result of each sub-comparison. Moreover, in case of equal votes, we compare the *summation* of the sub-similarities to remove the decision ambiguity. At this stage, we are able to compare different Android packages and make a decision on the most similar package to a given one. In what follows, we propose two approaches to malware detection.

Peer matching approach

In the *peer-matching* approach, ROAR uses the most straightforward technique to detect malware, i.e., query the fingerprints database to find the most similar fingerprint. To detect Android malware variation, we build a *malware fingerprints* database by computing APK-DNA for known Android malware. As shown in Fig. 6, for each new malware, we compute its APK-DNA and add it to the database.

To attribute the malware family to a new app, we first compute the similarity between the malware fingerprint and each entry in the database of known malware fingerprints, as depicted in Fig. 6. To this end, we use the *bitwise Jaccard* similarity, presented in Section Fingerprint similarity computation, to produce a set of sub-similarity values, i.e., *composed similarity*. Afterwards, to compare

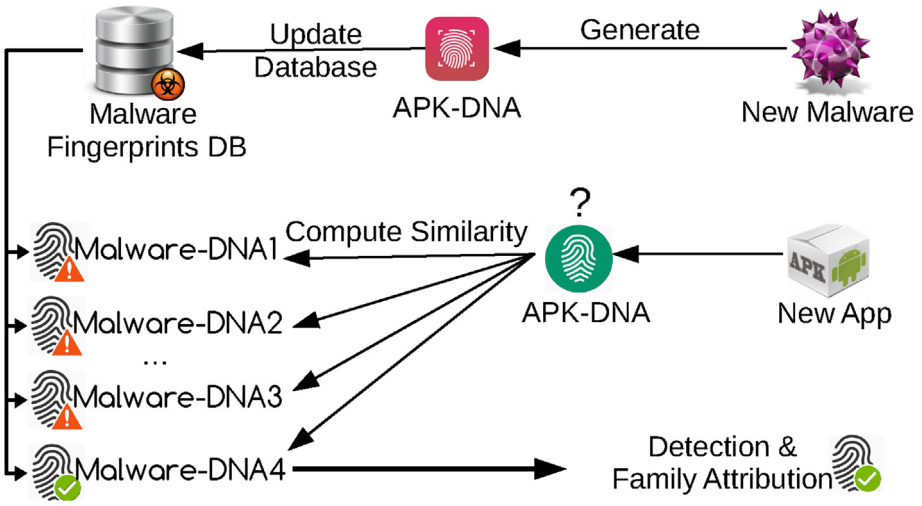


Fig. 6. Malware detection using peer-matching approach.

between *composed* similarity values, we use the previously presented *peer-voting* technique. The entry with the highest similarity, in case it exceeds a threshold, determines the malware family. In the current implementation, we use a static threshold. *Peer-matching* is a simple approach for malware detection and family attribution. However, this simplicity comes with a cost, as we will discuss in Section Discussion.

Family-fingerprinting approach

In this approach, some extra steps are needed to build a second database for family fingerprints. The target is to reduce the number of database entries needed to compare with in order to fingerprint an Android malware (see Fig. 7). For this reason, we propose a fuzzy hashing fingerprint for a malware family. The aim is to leverage this family fingerprint for malware detection. The idea is to build a database of family fingerprints from known Android malware samples, and use this database for similarity computation for unknown malware apps. The actual size of the family-fingerprints database is limited by the number of malware families. Notice that the fingerprint structure for a malware family is the same as for a single malware, i.e., metadata, binary, and assembly family sub-fingerprints.

Algorithm 4: Family Fingerprint Computation

```
input  : Malware Family X Fingerprints: Set
output: Family X Fingerprint: FP_X
FP_X = new bitvector[Zeros];
for fprint in Fingerprints do
    FP_X{meta} = FP_X{meta} or fprint{meta};
    FP_X{bin} = FP_X{bin} or fprint{bin};
    FP_X{asm} = FP_X{asm} or fprint{asm};
end
```

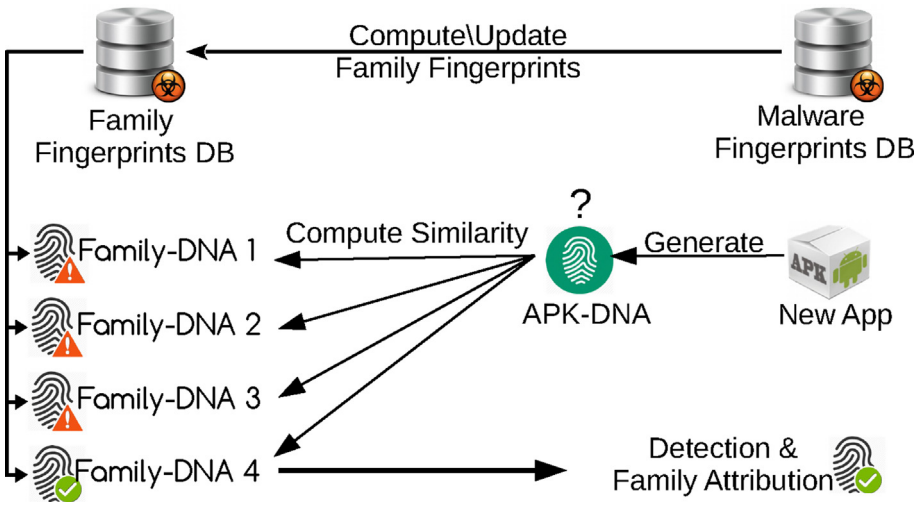


Fig. 7. Malware detection using family-fingerprint approach.

Algorithm 4 depicts the computation of the family fingerprint based on the underlying content sub-fingerprints. First, the fingerprint is initialized to zeros (each content sub-fingerprint). Afterwards, the fingerprint is generated by implying the logical *OR* of the current value of the family fingerprint with a single malware fingerprint. Note that each content sub-fingerprint is computed separately. This operation is applied on all malware samples in the database. After computing the fingerprints from known malware samples, we store them in a *family-fingerprints* database, which is used for detection and family attribution. The detection process is composed of several steps. First, for a given Android package, we generate its fingerprint as described in Section Fingerprint similarity computation. Then, we compute the similarity between this fingerprint and each family fingerprint in the database. The family with the highest similarity score will be chosen as the family of the new app if the similarity value is above a defined threshold. In the current implementation, we use a static threshold, which is only applied on the *summation* part of the composed similarity.

Experimental results

In this section, we start by describing ROAR implementation and the testing setup, including the dataset and the accuracy measurement techniques. Afterwards, we present the achieved results in terms of accuracy for both approaches that are adopted in ROAR framework, namely family-fingerprinting and peer-matching.

Implementation

Both ROAR approaches, namely family-fingerprinting and peer-matching, have been implemented and evaluated separately. We implemented ROAR using *Python* programming language and *bash* command line tools. As mentioned in Section APK-DNA fingerprint generation, we compress the content of the packages to increase the entropy. In the current implementation, *Gzip* compression tools have been chosen. For generating binary N-grams, we used *xxd* tool to convert the package content into a sequence of bytes. In addition, we use a set of command tools such as *awk* and *grep* to filter the results. Regarding reverse engineering of the *Dex* byte-code, we use *dexdump*, a tool that comes with Android SDK. The generated assembly is filtered using the common Unix tools. To extract the permission from *AndroidManifest.xml*, we first convert the binary XML to a readable format using *aapt*, a tool provided by Android SDK. Then, we parse the produced XML using standard Python XML parsing library.

Testing setup

Our dataset contains 928 malware samples from Android Malware Genome Project (Zhou and Jiang, 2012; Android Malware Genome Project, 2015). For the purpose of evaluation, we selected malware families with many samples since some malware families in Android Malware Genome Project (Android Malware Genome Project, 2015) contain only few samples (some families have only one sample), as depicted in Table 1. Clearly, by filtering out other families that do not have enough malware samples, we may miss the detection of these malware families. In our experiments, we particularly target malware variations, where the adversary repacks the original malware into different packages in order to deceive the mobile anti-malware and the evaluation process of Android app stores. For the purpose of evaluation, we decide to have the same size for each malware family so that all families will be equally represented in the fingerprints database. Having a balanced malware dataset is important for the evaluation of the *family-fingerprinting* approach since an unbalanced dataset could cause unbalanced density in the generated family APK-DNA. This choice has affected the detection and malware family attribution results. As such, in each iteration of the evaluation, we randomly sample 46 Android malware from the top families (any family with population greater than 46) in the dataset (Android Malware Genome Project, 2015) (**DroidDreamLight** family has the least malware in our testing dataset with 46 samples).

For each evaluation benchmark and from the balanced dataset, we randomly sample 70% of each family from the dataset to build the fingerprints database. The rest of the dataset (30%) is used for the evaluation of ROAR approaches and sub-fingerprints. Notice that the random sampling is done for every benchmark evaluation. Accordingly, we repeat the evaluation three times. The final result is the average of the three evaluation results. In addition to the known malware samples, we use benign Android applications in each evaluation. These apps have been downloaded from Google play pseudo-randomly without considering the popularity of the app, as shown in Table 1. For each evaluation, we randomly sample benign apps and include them in the testing. For each benchmark, we measure the number of *true positives* (TP), *false negatives* (FN), and *false positives* (FP) and compute the confusion matrix. Afterwards, we summarize the results by computing the general *precision* and *recall* using the formulas in Equations (3) and (4). Note that the highest the precision is, the fewer *false positives* we get in the matching. On the other hand, the highest the recall is, the fewer are the *false negative* results. The ideal result should have both high precision and high recall. To this end, the *F1-Score* is generally used to have the advantages of both

Table 1
Evaluation Malware dataset.

#	Malware Family/Apps	Number of samples
0	AnserverBot	187
1	KMin	52
2	DroidKungFu4	96
3	GoldDream	47
4	Geinimi	69
5	BaseBridge	122
6	DroidDreamLight	46
7	DroidKungFu3	309
8	Benign Apps	100

precision and *recall* with one single value. As shown in Formula 5, the *F1-score* is the *harmonic mean* of the precision and the recall. In addition, we compute the *confusion matrix* in the benchmark to determine the *false negatives* (FN) and *false positives* for a specific family.

$$Prescision = \frac{TP}{TP + FP} \tag{3}$$

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

$$F1 - Score = 2 \times \frac{Precision \times Recall}{Prescision + Recall} \tag{5}$$

ROAR evaluation results

In this section, we present the evaluation results of ROAR framework. Each approach has been separately evaluated. The results are presented using *F1-Score*, *Precision* and *Recall*. The approach has been evaluated multiple times using different fingerprint setups, i.e., combinations of sub-fingerprints, which are used to compute the similarities using *peer-voting* technique. Furthermore, we present a comparison between the proposed *peer-voting* similarity technique and a merged fingerprint similarity.

Confusion matrix description

In addition to the previous evaluation metrics, we also use the confusion matrices in each evaluation, as shown in Figs. 8 and 9. Each confusion matrix is a square table, where the number of rows and columns are respectively the malware families and the benign apps following the same order as in Table 1. The columns and rows from 0 to 7 are respectively the malware families, **AnserverBot, KMin, DroidKungFu4, GoldDream, Geinimi, BaseBridge, DroidDreamLight**, and **DroidKungFu3**, and the column and row 8 represent the benign apps. The interpretation of the confusion matrix results is related to the intensity of the color in its *diagonal*. The more the color is solid in the *diagonal*, the higher and the more accurate are the results of the evaluation (*true positive*). The intensity of the color of a given cell in the confusion matrix represents the number of the malware/apps that have been assigned to this cell according to the color bar. However, the less solid is the color in the *diagonal* and the more solid in the other cells, the less accurate is the result.

False negative

For any row from 0 to 7 (i.e., malware family), there is a missed malware family attribution if we have a gray color in the other cells of the same row. Even though we missed the family attribution, we still detect the maliciousness of the app. However, a gray cell in column 8 (benign apps) means that we missed both the detection and the family attribution (*detection false negative*).

False positive

In the benign apps row (row number 8), a gray color in the malware cells indicates that there is a *false positive*. In other words, there are benign apps that have been detected as malicious. The number of *false positives* could be measured using the intensity of the color according to the color bar.

Family-fingerprinting results

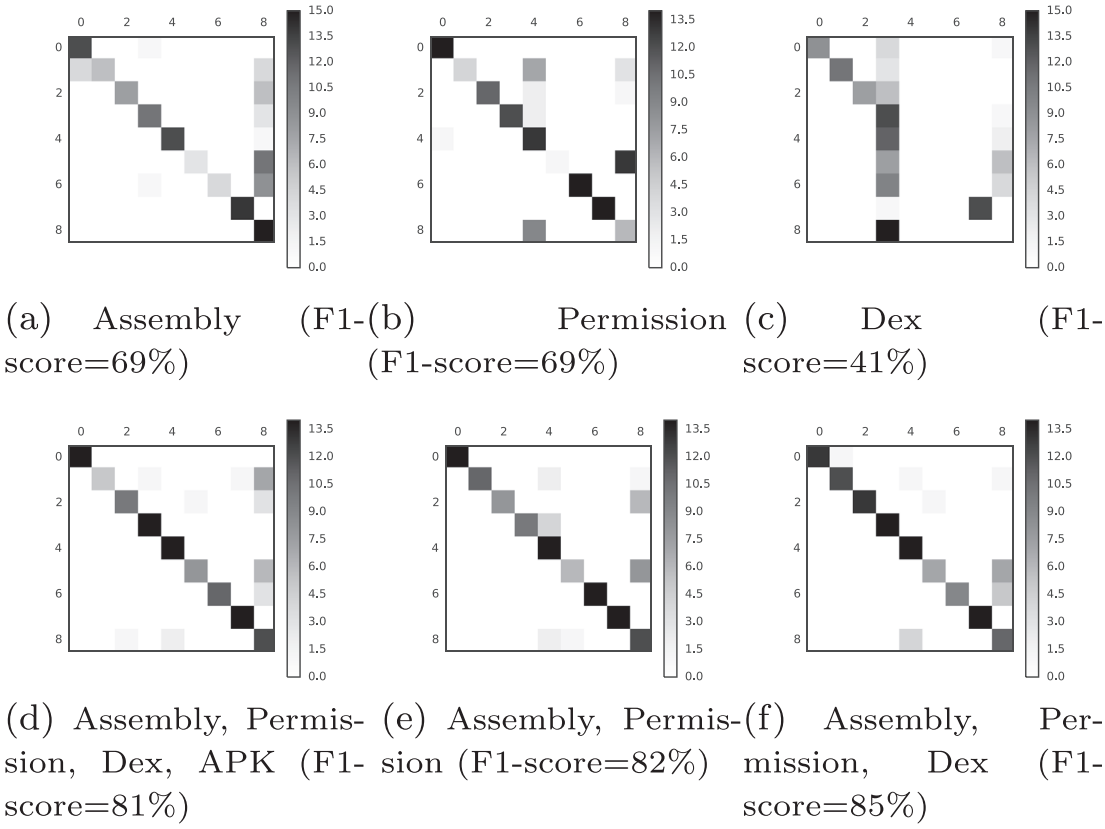
As depicted in Table 2, the F1-score, precision, and recall of the *family-fingerprinting* approach vary according to the fingerprint setup. We evaluated the approach for each content type separately, i.e., *assembly*, *permission*, and *dex* files, so that we can clearly see the impact of each component in the final fingerprint. Both *assembly* and *permission* types have shown more accurate results compared to *dex* type. Specifically, the *permission* has shown a very promising result, as shown in Table 2. This supports the impact of the metadata in Android malware detection. We believe that investigating other metadata could result in a higher accuracy. Another surprising result is *APK*, which has shown a poor accuracy value (under 40%). The learned lesson is that applying the fuzzy fingerprinting (including *ssdeep*) in the whole package could deceive the malware investigation using fuzzy matching. The confusion matrix for each setup shows a more granular view of the result, as shown in Fig. 8, where the indexes are the malware families (Table 1). On the other hand, the combination setups have shown accurate results for most combinations. We depict three sub-fingerprints, which correspond to the best results. Note that the setup composed of *assembly*, *permission*, and *dex* byte-code shows the highest *F1-Score*.

Peer-matching results

The *peer-matching* approach shows a higher F1-score, precision, and recall for all the setups compared to the *family-fingerprinting* one, as shown in Table 3. This can be clearly seen in the confusion matrices in Fig. 9. In contrast with the previous results, the *dex* byte-code shows a higher *precision* than *assembly* and *permission*, but it is still lower in both *recall* and *F1-score*. The setup combination (*assembly*, *permission*) has the highest accuracy in the *peer-matching* approach. As such, by using only two content categories, metadata *permission* and *assembly* instruction sequences, we achieved a very promising detection rate, especially that the computation of these sub-fingerprints is light and simple compared to state-of-the-art fingerprint hashing techniques.

Peer-voting vs merged fingerprints

As presented in Section Peer fingerprint voting technique, the most obvious technique to deal with multiple sub-fingerprints is to merge all of them (*merged fingerprint*). However, we propose *peer-voting* technique to compare between multiple sub-fingerprints. To test the proposed technique, we evaluate it against the *merged fingerprint* for the same fingerprint setup. As shown in Table 4, the *peer-voting* technique has shown a higher accuracy than the merging one. A more clear view of the result can be seen in the confusion matrix in Fig. 10.



(a) Assembly (F1-score=69%) (b) Permission (F1-score=69%) (c) Dex (F1-score=41%)

(d) Assembly, Permission, Dex, APK (F1-score=81%) (e) Assembly, Permission (F1-score=82%) (f) Assembly, Permission, Dex (F1-score=85%)

Fig. 8. Confusion Matrices of Family-Fingerprint Approach for each Fingerprint Setup.

Discussion

Similarity computation based on *family-fingerprinting* approach is bounded by the number of families, which is a way less than the number of malware samples. Consequently, this approach is more scalable compared to the *peer-matching* one. However, the *family-fingerprinting* approach suffers from two main drawbacks. First, its design is more complex compared to *peer-matching* since it requires an extra step to build the family-fingerprint database, even though we do not build the database for every new matching. Second, because of the rapid change of the malware database, we constantly need to update the family-fingerprint database with the new malware fingerprint features. On the contrary, the *peer-matching* approach does not require an update since the new malware fingerprint is directly inserted. Regarding the *peer-matching* approach, it mainly suffers from scalability issues since for each new malware, all the entries in the database need to be matched. Therefore, the query latency is related to the number of malware fingerprints in the database. However, in terms of accuracy, the *peer-matching* approach performs better than the *family fingerprinting* one. Concerning APK-DNA fingerprint, the average computation of APK-DNA is 15 s with medium size APK using only one core of the CPU. Finally, regarding the detection threshold, we used a fixed threshold to decide between malicious and benign apps based on our experimental results. Once an app is detected as malicious, it is assigned to the malware class with the highest similarity.

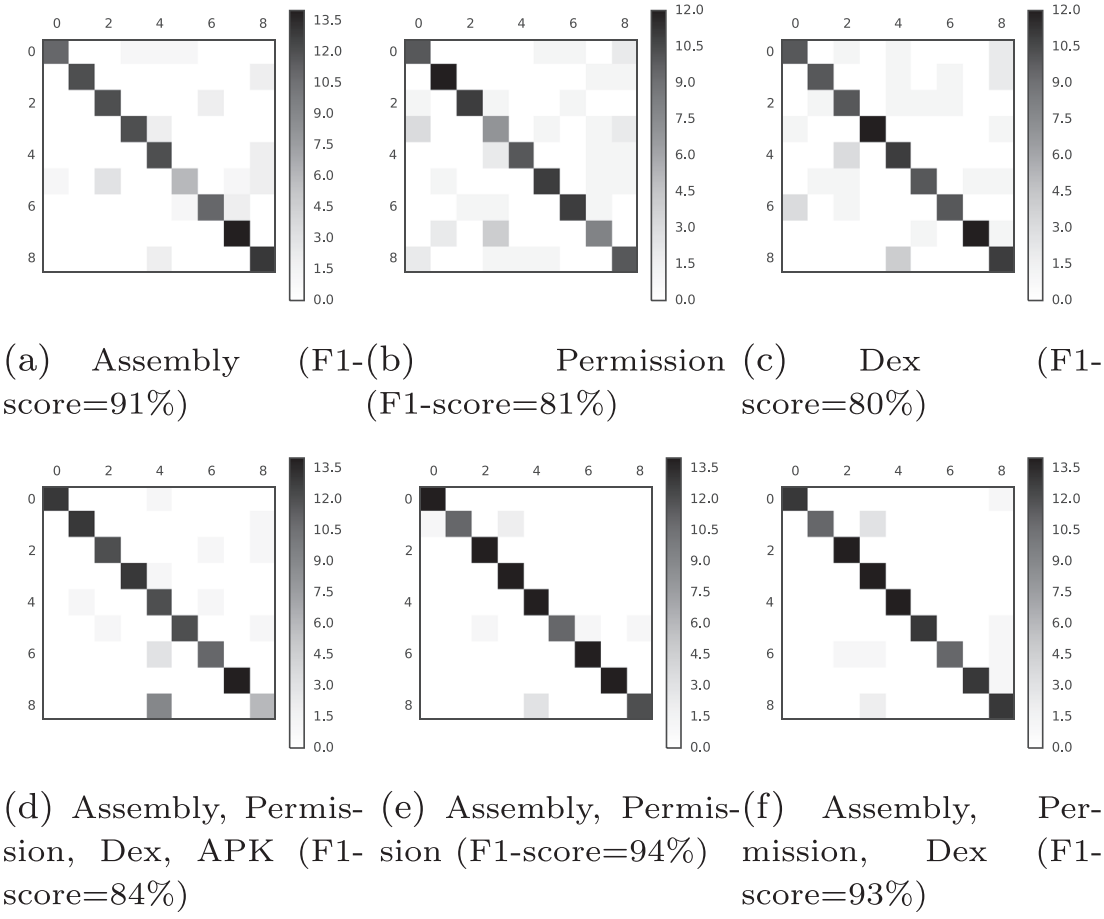
Limitations and future work

With the current implementation of ROAR and its fingerprinting technique, we cannot detect new Android malware since our main goal is to detect variations of known malware. Naturally, if the adversary needs to develop new malware or use very sophisticated repacking techniques, this malware might not be flagged. However, this will dramatically increase the cost for the attacker to repack known malware. Moreover, as presented in Section Discussion, the latency of the *peer-matching* approach is higher compared to the *family-fingerprinting* approach, and it linearly increases with the number of fingerprints in the database. Furthermore, in the current implementation of the proposed fingerprint, we did not include the *native library*, a part of the binary sub-fingerprint. We plan to consider the native code fingerprint in future work, where we focus on Android malware that uses C/C++ native code.

Related work

Signature-based malware detection

Li et al., (2015) evaluated the effectiveness of fuzzy hashing for clustering malware and proposed a new block-



(a) Assembly (F1-score=91%) (b) Permission (F1-score=81%) (c) Dex (F1-score=80%)

(d) Assembly, Permission, Dex, APK (F1-score=84%) (e) Assembly, Permission (F1-score=94%) (f) Assembly, Permission, Dex (F1-score=93%)

Fig. 9. Confusion Matrices of Peer-Matching Approach for each Fingerprint Setup.

Table 2
Accuracy results of the family-fingerprinting approach (The best fingerprint setup is in bold).

Fingerprint setup	F1-Score	Precision	Recall
Assembly	69%	88%	68%
APK	33%	36%	32%
Permission	69%	84%	70%
Dex	41%	46%	43%
Assembly, Permission, Dex, APK	81%	88%	80%
Assembly, Permission	82%	88%	81%
Assembly, Permission, Dex	**85%**	**89%**	**84%**
Best Fingerprint Setup	**85%**	**89%**	**84%**

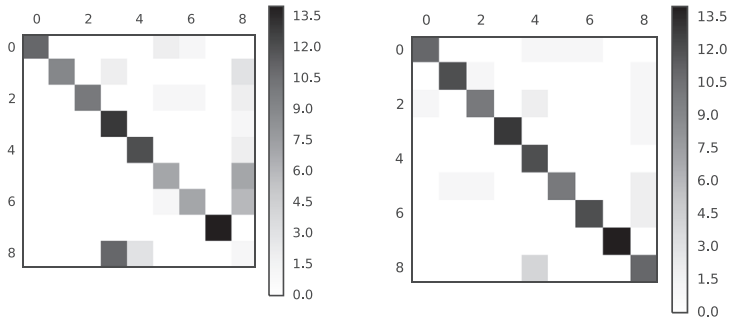
Table 3
Accuracy result of peer-matching approach (The best fingerprint setup is in bold).

Fingerprint setup	F1-Score	Precision	Recall
Assembly	91%	91%	90%
Apk	46%	48%	44%
Permission	81%	82%	80%
Dex	80%	90%	84%
Assembly, Permission, Dex, APK	84%	91%	81%
Assembly, Permission, Dex	93%	94%	93%
Assembly, Permission	**94%**	**95%**	**94%**
Best Fingerprint Setup	**94%**	**95%**	**94%**

Table 4
Accuracy result of ROAR using merged fingerprint (The best fingerprint setup is in bold).

Fingerprint setup	F1-Score	Precision	Recall
Merged in Family-Approach	72%	84%	72%
Peer-Voting in Family-Approach	**85%**	**89%**	**84%**
Merged in Peer-Approach	86%	87%	86%
Peer-Voting in Peer-Approach	**94%**	**95%**	**94%**

based distance computation algorithm. DroidMOSS (Zhou et al., 2012a) generates app-specific fingerprints to detect app modifications and computes similarity scores using edit distance. Crussell et al., (2013) proposed a system based on *min-hashing* and *LSH* techniques to improve the matching time. Hanna et al., (2012) proposed the use of Delvik VM assembly op-codes to compute a list of features and the similarity between inputted apps. Juxtapp (Hanna et al., 2013) uses feature hashing on the opcode sequence to determine if an app is malicious, contains copies of buggy or plagiarized code. Nayak et al., (2014) proposed a fuzzy approach to classify Android malware into: root exploit, information steal, premium SMS, trojan, and benign. Other signature-based contributions (Faruki et al., 2013, 2015; Ali-Gombe et al., 2015; Lindorfer et al., 2014a;



(a) Merged Fingerprint in Family-Fingerprint Approach (F1-score=72%)

(b) Merged Fingerprint in Peer-Matching Approach (F1-score=86%)

Fig. 10. Confusion matrices of ROAR approaches using merged fingerprint.

Feng et al., 2014; Zhou et al., 2012b, 2013; Chen et al., 2015) have been proposed using different approaches. Most of these solutions consider only one content category of APK, for example, Juxtapp focuses on the assembly extracted from the DEX byte-code, (Crussell et al., 2013) considers malware's *code section*, (Ali-Gombe et al., 2015) is limited to classes.dex file, and (Crussell et al., 2013) does not consider third-party libraries. In contrast, our solution considers multiple content categories as well as the metadata, which allows APK-DNA to have much coverage on the static characteristics of the malicious apps.

Runtime-based malware detection

Vidas and al (Vidas and Christin, 2014; Vidas et al., 2014) explored various emulator and dynamic analysis techniques for Android systems based on differences in behavior, performance, components and system design. ANDRUBIS (Lindorfer et al., 2014b) analyzed over 1,000,000 Android apps based on extracted features and observed behaviors during runtime. Among many other runtime-based contributions, we can cite (Rastogi et al., 2013; Yan and Yin, 2012; Spreitzenbarth et al., 2015; Enck et al., 2014; Zhang et al., 2013; Zhang and Yin, 2014). Researchers in Ref. (Sufatrio et al., 2015) highlighted some of the challenges faced by dynamic analysis techniques on Android apps, such as, the need to simulate system events and the user's GUI responses, time limit on how an app can be executed and observed, detection of the presence of Android virtualization or emulation systems by malicious applications, in addition to scalability requirements of such detection systems to evaluate an enormous amount of available apps.

Android malware classification

Peng et al., (2012) proposed a classification and probabilistic models for ranking risks for Android apps. DroidSIFT (Zhang et al., 2014) is a semantic-based system that classifies Android malware using dependency graphs. DroidAPIMiner (Aafer et al., 2013) extracts Android malware features at the API level and provides robust and light classifiers to mitigate malware installations. Drebin (Arp et al., 2014) detects malicious applications based on Android permissions and sensitive APIs. The main difference with our work is that most existing approaches are based on machine learning techniques to generate models for malware detection and attribution. In contrast, we generate fuzzy fingerprints to fill the gap of existing signature techniques, such as MD5, ssdeep, etc. Our technique could be used in complementary to existing machine learning solutions where APK-DNA could be used as a first filtration phase.

Conclusion

In this paper, we have presented a comprehensive fuzzy fingerprinting approach for investigating Android malware variations. To this end, we have proposed APK-DNA, a novel fingerprint that captures not only the binary of the *APK* file, but also both its structure and semantics. This is of paramount importance since it allowed the proposed fingerprint to be highly resistant to app changes, which is a significant advantage compared to traditional fuzzy hashing techniques. Moreover, we have leveraged the proposed APK-DNA fingerprint to design and implement a novel framework for Android malware detection, following two different approaches, namely *family-fingerprinting* and *peer-matching*. In addition to detecting malware variations, ROAR framework is also able to attribute the family of the detected malware. The evaluation of ROAR demonstrated very promising results, with high accuracy in terms of both malware detection and family attribution.

Acknowledgments

The authors would like to thank the anonymous reviewers and Timothy Vidas for their insightful comments that allowed us to significantly improve this paper.

Appendix A. Supplementary data

Supplementary data related to this article can be found at http://dx.doi.org/10.1016/j.diin.2016.04.013.

References

Aafer Y, Du W, Yin H. Droidapiminer: mining api-level features for robust malware detection in android. In: Security and privacy in communication networks - 9th international ICST conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013; 2013. p. 86–103. Revised Selected Papers.

Ali-Gombe A, Ahmed I, Richard III GG, Roussev V. OpSeq: android malware fingerprinting. In: Proceedings of the 5th program protection and reverse engineering workshop, PPREW-5, ACM, New York, NY, USA; 2015. 7:1–7:12.

Android Malware Genome Project (2015). URL http://www.malgenomeproject.org/.

Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K. Drebin: effective and explainable detection of android malware in your pocket. In: NDSS, the internet society; 2014.

Breitinger F, Baier H. Similarity preserving hashing: eligible properties and a new algorithm mrsh-v2. Digit Forensics Cyber Crime 2013:167–82.

Breitinger F, Astebl K, Baier H, Busch C. mvHash-B – a new approach for similarity preserving hashing. In: IT security incident management and it forensics (IMF), 2013 seventh international conference on; 2013. p. 33–44.

Chen J, Alalfi MH, Dean TR, Zou Y. Detecting android malware using clone detection. J Comput Sci Technol 2015;30(5):942–56.

Cisco. Cisco 2014 annual security report. 2014. p. 1–81. http://tinyurl.com/lfvm2ut.

Crussell J, Gibler C, Chen H. Andarwin: scalable detection of semantically similar android applications. In: Computer security - ESORICS 2013-18th European symposium on research in computer security, Egham, UK, September 9-13, 2013. Proceedings; 2013. p. 182–99.

Department of Defense Computer Forensics Lab (DCFL). dcfldd - latest version 1.3.4–1. 2016. http://dcfldd.sourceforge.net/.

Enck W, Gilbert P, Chun B, Cox LP, Jung J, McDaniel P, et al. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. Commun ACM 2014;57(3):99–106.

Ericsson. Ericsson mobility report - on the pulse of the networked society. 2013. p. 4–7. http://tinyurl.com/zwoh8c6.

Faruki P, Laxmi V, Ganmoor V, Gaur MS, Bharmal A. Droidolytics: robust feature signature for repackaged android apps on official and third party android markets. In: Advanced computing, networking and security (ADCONS), 2013 2nd international conference on; 2013. p. 247–52.

Faruki P, Laxmi V, Bharmal A, Gaur MS, Ganmoor V. AndroSimilar: robust signature for detecting variants of Android malware. J Inf Secur Appl 2015;22:66–80.

Feng Y, Anand S, Dillig I, Aiken A. Apposcopy: semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, (FSE-22), Hong Kong, China, november 16-22, 2014; 2014. p. 576–87.

Figuerola CG, Díaz RG, Berrocal JLA, Rodríguez ÁFZ. Web document duplicate detection using fuzzy hashing. In: Trends in practical applications of agents and multiagent systems, 9th international conference on practical applications of agents and multiagent systems (PAAMS 2011) workshops; 2011. p. 117–25.

Getting Started with the NDK (2016). URL http://tinyurl.com/pah7pab.

Hanna S, Huang L, Wu EX, Li S, Chen C, Song D. Juxtapp: a scalable system for detecting code reuse among android applications. In: Detection of intrusions and malware, and vulnerability assessment – 9th international conference, DIMVA 2012, Heraklion, crete, Greece, July 26-27, 2012; 2012. p. 62–81. Revised Selected Papers.

Hanna S, Huang L, Wu E, Li S, Chen C, Song D. Juxtapp: a scalable system for detecting code reuse among android applications. In: Proceedings of the 9th international conference on detection of intrusions and malware, and vulnerability assessment, DIMVA'12. Berlin, Heidelberg: Springer-Verlag; 2013. p. 62–81.

Jang J, Brumley D. BitShred: feature hashing malware for scalable triage and semantic analysis. Security 2011:309–20.

Kornblum JD. Identifying almost identical files using context triggered piecewise hashing. Digit Investig 2006;3(Supp. 1):91–7.

Li Y, Sundaramurthy SC, Bardas AG, Ou X, Caragea D, Hu X, et al. Experimental study of fuzzy hashing in malware clustering analysis. In: Proceedings of the 8th USENIX conference on cyber security experimentation and test, CSET'15. Berkeley, CA, USA: USENIX Association; 2015. 8–8.

Lindorfer M, Volanis S, Sisto A, Neugschwandtner M, Athanasopoulos E, Maggi F, et al. Andradar: fast discovery of android applications in alternative markets. In: Detection of intrusions and malware, and vulnerability assessment - 11th international conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings; 2014. p. 51–71.

Lindorfer M, Neugschw M, Weichselbaum L, Fratantonio Y, Veen VVD, Platzer C. Andrubis- 1,000,000 apps later: a view on current android malware behaviors. In: Proceedings of the 3rd international workshop on building analysis datasets and gathering experience returns for security, BADGERS 2014, Wroclaw, Poland, September 11th, 2014; 2014.

Masud MM, Khan L, Thuraisingham B. A scalable multi-level feature extraction technique to detect malicious executables. In: IEEE int. Conf. Commun; 2007. p. 1443–8.

Nayak A, Yen K, Pons A. Fuzzy logic based android malware classification approach. Int J Comput Netw Secur 2014;24(1):8.

Peng H, Gates C, Sarma B, Li N, Qi Y, Potharaju R, et al. Using probabilistic generative models for ranking risks of android apps. In: Proceedings of the 2012 ACM conference on computer and communications security, CCS '12. New York, NY, USA: ACM; 2012. p. 241–52.

Rastogi V, Chen Y, Enck W. AppsPlayground: automatic security analysis of smartphone applications. In: Proceedings of the third ACM conference on data and application security and privacy, CODASPY '13. New York, NY, USA: ACM; 2013. p. 209–20.

A. Tridgell, P. Mackerras, The rsync algorithm, Joint Computer Science Technical Report Series TR-CS-96-0. URL http://tinyurl.com/zm5g5jw.

V. Roussev, An evaluation of forensic similarity hashes, Digit Investig 8 (Suppl.).

Roussev V. Data fingerprinting with similarity digests. In: Advances in Digital forensics VI - sixth IFIP WG 11.9 international conference on Digital Forensics, Hong Kong, China, January 4-6, 2010; 2010. p. 207–26. Revised Selected Papers.

sdhash (2016). URL http://sdhash.org.

Spreitzenbarth M, Schreck T, Echtler F, Arp D, Hoffmann J. Mobile-Sandbox: combining static and dynamic analysis with machine-learning techniques. Int J Inf Sec 2015;14(2):141–53.

Sufatrio, Tan DJJ, Chua T, Thing VLL. Securing android: a survey, taxonomy, and challenges. ACM Comput Surv 2015;47(4):58.

Vidas T, Christin N. Evading android runtime analysis via sandbox detection. In: 9th ACM symposium on information, computer and communications security, ASIA CCS '14, Kyoto, Japan - June 03-06, 2014; 2014. p. 447–58.

Vidas T, Tan J, Nahata J, Tan CL, Christin N, Tague P. A5: automated analysis of adversarial android applications. In: Proceedings of the 4th ACM workshop on security and privacy in smartphones & mobile devices, SPSM@CCS 2014, Scottsdale, AZ, USA, November 03-07, 2014; 2014. p. 39–50.

Wang C, Shieh SW. DROIT: dynamic alternation of dual-level tainting for malware analysis. J Inf Sci Eng 2015;31(1):111–29.

Yan LK, Yin H. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX conference on security symposium, Security'12. Berkeley, CA, USA: USENIX Association; 2012. 29–29.

Zhang M, Yin H. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In: 9th ACM symposium on information, computer and communications security, ASIA CCS '14, Kyoto, Japan - June 03-06, 2014; 2014. p. 259–70.

Zhang Y, Yang M, Xu B, Yang Z, Gu G, Ning P, et al. Vetting undesirable behaviors in android apps with permission use analysis. In: 2013 ACM SIGSAC conference on computer and communications security, CCS'13, Berlin, Germany, November 4-8, 2013; 2013. p. 611–22.

Zhang M, Duan Y, Yin H, Zhao Z. Semantics-aware android malware classification using weighted contextual api dependency graphs. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, CCS '14. New York, NY, USA: ACM; 2014. p. 1105–16.

Zhou Y, Jiang X. Dissecting android malware: characterization and evolution. In: Proc. - IEEE symposium on security and privacy (4); 2012. p. 95–109.

Zhou W, Zhou Y, Jiang X, Ning P. Detecting repackaged smartphone applications in third-party android marketplaces. In: Second ACM conference on data and application security and privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012; 2012. p. 317–26.

Zhou Y, Wang Z, Zhou W, Jiang X. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: 19th annual network and distributed system security symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012; 2012.

Zhou W, Zhou Y, Grace MC, Jiang X, Zou S. Fast, scalable detection of piggybacked mobile applications. In: Third ACM conference on data and application security and privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013; 2013. p. 185–96.