DroidKin: Lightweight Detection of Android Apps Similarity

Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani

Information Security Centre of Excellence,
Faculty of Computer Science, University of New Brunswick, Canada
`hugo.gonzalez,natalia,ghorbani@unb.ca`

Abstract. The appearance of the Android platform and its popularity has resulted in a sharp rise in the number of reported vulnerabilities and consequently in the number of mobile threats. Leveraging openness of Android app markets and the lack of security testing, malware authors commonly plagiarize Android applications (e.g., through code reuse and repackaging) boosting the amount of malware on the markets and consequently the infection rate.
In this paper, we present DroidKin, a robust approach for the detection of Android apps similarity. Based on a set of characteristics derived from binary and meta data accompanying it, DroidKin is able to detect similarity among applications under various levels of obfuscation. DroidKin performs analysis pinpointing similarities between applications and identifying their relationships. We validated our approach on a set of manually prepared Android applications and evaluated it with datasets made available by three recent studies: The Android Malware Genome project, Drebin, DroidAnalytics. This data sets showed that several relations exists between the samples. Finally, we performed a large-scale study of over 8,000 Android applications from Google play and Virus Total service.

Key words: Android, malware, similarity

1 Introduction

An appearance of a new Android platform and its popularity has resulted in a sharp rise in the number of reported vulnerabilities and consequently in the number of threats. This unprecedented growth has swiftly attracted an attention of a security community resulting in a number of security solutions for malware detection, response and analysis.

The lack of suitable datasets has quickly proved to be the major hindrance for research efforts in the field. To remedy the situation a number of studies ventured to generate several malware data sets [13, 16, 36], some of which quickly became benchmarks for malware analysis and evaluation.

There are generally two criteria that are considered for inclusion of a malware sample into a data set: uniqueness of a sample and its ability to represent a family of interest. Traditionally cryptographic hash values have served as unique identifiers (i.e., fingerprints) of malware samples. MD5 and SHA have been the most common hash functions usually employed for this purpose. In spite of their wide spread implementation, the use of hash values has been seen as very restrictive mostly due to its inability to allow insignificant sample modifications. Such modifications often come from application of various obfuscation techniques varying from simple repackaging to encryption that change the form of a malware sample (i.e., binary) while retaining the same functionality.

Detection of mobile app repackaging, as well as general detection malware apps have been extensively studied in the last several years. RiskRanger [20], DroidRanger [37], Drebin [7], DroidScope [31] are among general detection methods that are able to pinpoint malicious behavior either through a dynamic analysis of app's run-time behavior or through its static analysis. Although many of these methods offer good accuracy and scalability, all of them focus on producing a binary output generally indicating whether an app is benign or not. Several studies gave a deeper insight into possibly malicious apps introducing methods for detecting repackaged applications [35, 22]. Similarly to the general detection techniques, these methods are designed to indicate whether an app is repackaged or not.

Unfortunately, these methods are not sufficient for comprehensive evaluation and study of mobile malware for several reasons. Evaluating classification accuracy of any malware detection method requires a clear understanding of the data, i.e., samples' distribution across families, diversity of samples, their uniqueness and existence of duplications. Such transparent view of data is essential for accurate assessment of the method's performance. For example, this allows to understand whether current method performance is due to the majority of samples being descendants of the same original instance and essentially being identical in nature or it reflects method's true detection ability in a real world environment.

The ability to prepare such data set for evaluation requires the existence of suitable lightweight methods equipped with means to give a multidimensional view of sample's maliciousness. Most of the existing methods, however, do not indicate the reasoning behind their decisions, and similarly do not identify the relations between malicious apps [7]. They also employ sophisticated heuristics incurring run-time overhead [37] or requiring hand-crafted detection patterns [20], thus confining method's application to a certain (not always available) deployment environment.

In this paper, we develop a lightweight approach to identify Android apps similarity and infer their relationship to each other. More specifically, the proposed approach called DroidKin allows us to detect the existence of similarity between apps and understanding its nature, i.e., how and why the apps are related. This assessment is based on a static analysis of a set of characteristics gathered from application's package. To avoid pair-wise analysis of all apps, our approach employs a filtering stage that guides the similarity assessment process to only a subset of related applications. This efficiency enables a deeper analysis of selected apps providing more insight into their similarity relationships.

We validate our approach on a set of manually prepared Android applications. We further evaluate it with datasets made available by three recent studies: The Android Malware Genome project, Drebin, DroidAnalytics and performed a large-scale study of over 8,000 Android application from Google market and Virus Total repository.

The rest of the paper discusses the related works in Section 2, presents the details of the proposed approach in Sections 4, and validation and evaluation results in Sections 5 and 6. Section 8 concludes the paper.

2 Related Work

The past decade has been marked with extensive research in the area of mobile security. A broad study looking at a variety of mobile technologies (e.g., GSM, Bluetooth), their vulnerabilities, attacks, and the corresponding detection approaches was conducted by Polla et al. [23]. More focused studies surveying characteristics of mobile malware were offered by Alzahrani et al. [6], Felt et al. [16] and Zhou et al. [36].

With the recent burst of research interest in the area of Android device security, there have been a number studies focusing on mobile malware detection. These studies include detection of privacy violations during apps' runtime (TaintDroid [14], MockDroid [9], VetDroid [32]), unsafe permission usage [15, 8, 27, 12, 29] and security policy violations [24, 28]. All these techniques are designed for detection of specific violations that deem an app to be abnormal and potentially malicious. As such they are unsuitable for more general analysis of mobile apps for their uniqueness (e.g., detection of legitimate repackaged apps or malicious apps not requesting suspicious permissions).

There were a number of general studies offering methods for malicious app detection. These methods can be broadly divided into those focused on the detection prior to app installation (e.g., market analysis) and those that monitor app behavior directly on a mobile device.

Among the studies in the first group are RiskRanger [20], DroidRanger [37], DroidScope [31] that dynamically monitor mobile apps behavior in an isolated environment collecting detailed information that might indicate maliciousness of a sample. Similarly, DroidMat [30] and DroidAPIMiner [5] looked at identification of malicious apps using machine leaning techniques. Since these techniques are computationally expensive for a resource-constraint environment of a mobile platform, they are mostly intended for an offline detection. A number of studies introduced lightweight approaches to malware detection to be applied on a mobile device directly, among them is Drebin [7]. This approach employs static analysis in combination with machine learning to detect suspicious patterns in app behavior. Although Drebin aims to provide explainable results, it is able to give insight into malware uniqueness.

With a recent wave of cloned applications, a number of studies looked at the problem of apps similarity in mobile apps. A general overview of plagiarized apps was given by Gibler et al [19]. The majority of the existing methods look at the content of .dex files for app comparison. Juxtapp [21] evaluates code similarity based k-grams opcodes extracted from selected packages of the disassembled .dex file. The generated k-grams are hashed before they undergo a clustering procedure that groups similar apps together. Similarly, DroidMOSS [35] evaluates app similarity based of fuzzy hashes constructed based on a series of opcodes. Aside from opcodes other methods can be employed to fingerprint mobile apps. For example, AnDarwin [10], DNADroid [11], and PiggyApp [34] employ Program Dependance Graphs (PDG) that represent dependencies between code statements/packages. Potharaju et al. [26] computes fingerprints using several methods based on Abstract Syntax Tree (AST).

3 Background

An Android app is written in Java language and compiled into a .dex file that can be run by Dalvik virtual machine on an Android platform. The apps are packaged in an .apk file containing the executable .dex file; manifest.xml file that describes the content of the package including the permissions information; native code (optional) in form of executable or libraries that usually is called from the .dex file; the file with a digital certificate authenticating an author; and the resources that the app uses (e.g., image files, sounds). Each .apk file is annotated with additional information, so called meta-data, such as the app creation date and time, version, size, etc.

While the majority of the existing approaches are focusing their analysis on the .dex file, there are a number of other factors that need to be considered.

First of all, the digital certification plays an important role in Android apps. This is the only mechanism developed to attest the ownership of an app. The certificates are self-signed, i.e., no certificate authority (CA) is required. While the mechanism was originally meant to tie an author to an app, allowing a legitimate owner to issue new apps and update older versions under the same key, several problems have quickly surfaced. Through our preliminary analysis of Android markets, we discovered that the authors (both legitimate and malicious) tend to generate a new pair of private/public keys for each application. The value of the self-certification is also undermined through an extensive use of public keys made available for debugging purposes. Although these days the official Google play market bans apps signed with these keys, other markets do not seem to enforce this policy. Finally, the recently discovered master key vulnerability opened a new window allowing attackers to inject malware into legitimate apps without invalidating a digital certification [17]. These weaknesses challenge the legitimacy of using digital certificates for app authorship identification. In this light some of the methods designed to rely on the existence of the original app (determined based on the certification) for detection of plagiarized applications (e.g., [35]) require readjustment.

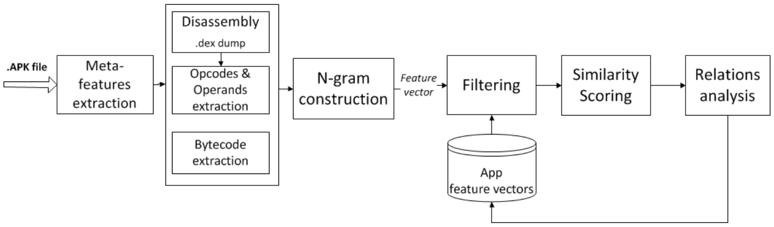

Fig. 1: The architecture of DroidKin

Another practice that have been gaining popularity in mobile apps is the use of external code, i.e., an additional code that is loaded and executed at runtime of an app [25]. This mechanism allows for the use of legitimate applications to load malicious functionality without requiring any modifications to the existing legitimate code. As such the original bytecode remains intact allowing the app to evade detection. Poeplau et al. [25] defined several techniques to load external code on a device: with the help of class loaders that enable the extraction of classes from files in arbitrary locations, through the package context that allows for the access of resources of other apps, through the use of native code, with the help of runtime method exec that gives access to a system shell, and through less stealthy installation of additional .apk files requested by a main .apk file.

Obfuscation Although code obfuscation prevails in desktop malware, mobile malware obfuscation is gaining popularity in mobile devices. Potharaju et al. [26] have defined two obfuscation levels: basic Level-1 obfuscation that includes renaming and removal of unused identifiers (e.g., class, variable, methods), and the more advanced Level-2 obfuscation that includes insertion of junk code.

Until now though the most common obfuscation method applied in practice was Level-1 [26]. Obfuscation gained popularity partially due to the wide spread of repackaged applications, as it allows effective prevention of piggybacking malicious payload into original apps [36]. In this work we experiment with various obfuscation methods and their impact on app similarity detection.

4 Approach design

The architecture of our system is in many respects dictated by the nature of work. Under the broader umbrella of similarity detection, we are focusing on detecting the presence of similarity and understanding its nature, i.e., how and why the apps are related. As such the proposed system encompasses three steps: *feature extraction*, *similarity assessment* and *relationship analysis* (Figure 1).

For each app requiring similarity analysis, the system derives relevant features and forms vectors that serve as a basis in the similarity assessment. Based on these feature vectors, potential candidates that might have some relations to a given app are identified and scored. Once the similarity is established, the nature of the relations between apps is derived as a combination of computed scores and participating features. Note that due to a diversity of obfuscating techniques for different apps different features may contribute to the presence and the extend of similarity. As a result the last step mainly focuses on examining all possible relations between similar apps prioritizing them based on the type of features involved.

4.1 Features

We extracted the following features that serve as unique characteristics of a given app:

- *Meta-information* that accompanies each .apk file (*META-INF* directory) and characterizes its content. These features can be broadly divided into two groups: *descriptive* features that include certificate serial number, the hash value (md5) for the .apk container and a .dex file (md5), and a list of internal files' names with corresponding hash values (md5); and *numerical* features such as a number of internal .apk, .zip, java, .jar, images, libraries and binary files found within a container .apk file; size of .apk and .dex files; number of files in .apk file.
- *N-grams* characterizing the .dex file. In the literature, malware analysis is typically conducted at bytecode and opcode levels. Opcodes are generally beneficial in representing low-level semantics of the code. Since extracting opcodes alone might abstract specific details describing a program control transfer or an arithmetical operation, opcodes are often enhanced with the corresponding operands. Bytecode is seen as the complete representation of the code at low-level. As such we experimented with opcode n-gram (with and without operands) and with bytecode n-grams.

The extracted features are abstracted in a feature vector composed of two parts corresponding to meta-information features and n-grams respectively.

4.2 Similarity Assessment

The similarity between apps is assessed in two stages: filtering stage and the similarity scoring stage.

Filtering The primary goal of the filtering process is to reduce the number of comparisons necessary to find the relationships between the analyzed app and the apps which information is already stored in a database. Filtering is mostly based on meta-data features guiding the analysis process to a reduced set of apps which require similarity scoring. The flow of this process is presented in Figure 2.

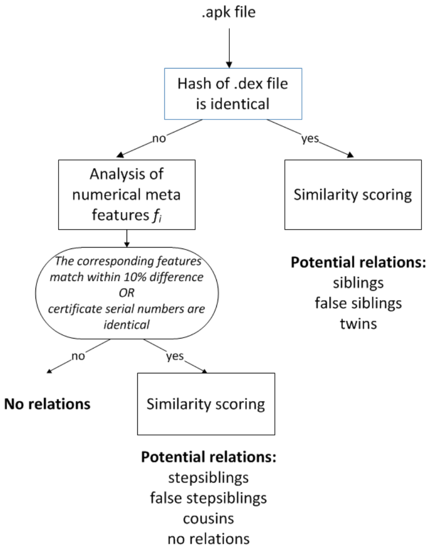

Fig. 2: Filtering process

Similarity scoring Once the set of potentially related apps is identified, the pairwise similarity between a given app and this set is calculated using a variation of a similarity measure, called *Simplified Profile Intersection (SPI)* proposed by [18]. The metric was proposed for evaluation of source code author profiles often limited in size and thus unable to offer a reliable estimate for distance metrics based on frequency analysis. Given *Simplified Profiles* SP_i and SP_j for authors i and j, the similarity distance, called *Similarity Profile Intersection* SPI is the size of intersection between these profiles. Formally, the normalized SPI is given as follows:

$$SPI = \frac{|SP_i \bigcap SP_j|}{max(|SP_i|, |SP_j|)} \tag{1}$$

In other words, the similarity between two apps is defined by the amount of commonalties existing in their profiles. Thus, the larger the size of intersection, the more similar two apps are.

In our context, we compute a normalized similarity distance between two apps using the descriptive meta features, $SPIf$ and the n-grams, $SPIng$. The similarity between meta features is computed separately for hashes $SPIf_h$ and file names $SPIf_n$ using formula (1). The resulting values are then combined (with a preference to similarity of files' content) as follows

$$SPIf = 60 \times SPIf_h + 40 \times SPIf_n$$

Similarity profile intersection based on the n-gram vectors, $SPIng_n$, is also calculated using formula (1). In this case, SP is represented by an n-gram

Table 1: Relationships' thresholds.

Twins:	$SPIng = 100\%$ and $SPIf > 95\%$
Siblings:	$SPIng = 100\%$ and $SPIf > 60\%$
False Siblings:	$SPIng = 100\%$ and $SPIf < 60\%$
Step siblings:	$SPIng > 60\%$ and $SPIf > 60\%$
False step siblings:	$SPIng > 60\%$ and $SPIf < 60\%$
Cousins:	$SPIng < 60\%$ and $SPIf > 60\%$

frequency vector of an app. To generate these frequency vectors, each app is disassembled/processed to extract unique opcodes, opcode/operand pairs or bytecode. Let Op represent this sequence of opcodes, then $SP = (f_m)_{1 \leq m \leq k}$, where f_m is the frequency (i.e., the number of occurrences) of opcode (or opcode/operand pair, bytecode) $o_m \in Op$.

The resulting $SPIf$ and $SPIng$ values are used to establish a relationship between a pair of apps.

4.3 Relationships

One of the goals of this study is to give deeper insight into the nature of apps similarity. To abstract the details while providing a better sense of related apps, we introduce the following definitions that outline relations between apps (in decreasing order of closeness):

- *Twins*: are the apps with almost identical content, but different hash of .apk file. Typically these are repackaged applications. The difference in packaging time or a lack of alignment results in different hash value.
- *Siblings*: have identical .dex file, but as opposed to twins they only have some of resource files in common (not all of them). This scenario is common for piggybacked apps that retain the same original code while adding malicious functionality, for instance, by loading external code through resources.
- *False siblings*: are sibling apps that do not have many resources in common.
- *Step siblings*: are even more distant from the twins. These apps share the majority of the .dex file and the majority of the resources indicating that although the content is likely to be plagiarized the app introduces additional functionality.
- *False stepsiblings*: are the apps that appear to be step siblings, but do not share many of the resources.
- *Cousins*: are defined as distant relatives that do not share common content with .dex file, they however, employ the majority of same resources.

The exact relationship between two apps is derived via a filtering process and analysis of similarity values $SPIf$ and $SPIng$. Through our preliminary experiments, we identified several similarity thresholds that maintain high accuracy while causing no false positives (Table 1).

5 Validation

The lack of comprehensive datasets has been repeatedly emphasized as a significant problem. Although several datasets were generated, the selection of samples was mostly done on the basis of hash uniqueness. As a result, none of these existing sets can serve as a 'ground truth' data for our experimentation purposes. To ensure a comprehensive evaluation of the proposed approach, we constructed a validation dataset with known relations between apps.

5.1 Data

For the validation dataset 72 unique Android apps were selected from different sources. We manually selected one sample from each family of the Android Malware Genome Project dataset [36], eight samples from the Android Malware Virus Share package [2], nine samples from the Virus Total [3] and five samples from the official Google Play market. Their uniqueness was verified through VirusTotal labeling and confirmed manually.

To investigate the impact of obfuscation on similarity detection, this set of unique apps underwent the following transformations:

Modification of .dex file:

- *Repackaging.* Using apk tool, we unpack the .apk file, to the smali presentation of the .dex file, then repackage the content back into an .apk file without modifying the content. This transformation alters the timestamp of all files and produces a new .dex file which results in step sibling relation between the original and repackaged version of the app.
- *Rerepackaging.* Using the same apk tool, we unpack the already repackaged file to the smali representation, and repackaged again. Similar to repackaging this transformation produces step sibling relations.
- *String url modifications.* Common well known urls such as google.com, bing.com, yahoo.com were kept intact, while the rest of the urls were replaced with randomly generated strings. Such modification changes the content of the .dex file producing step siblings.
- *Junk code insertion.* Using the apk tool, we unpack the .apk file to its smali representation of the .dex file, then we modify the code adding junk random code at the beginning of every public method. The junk code was designed to not do anything, and therefore did not change the functionality of the app. This transformation only affected the final .dex file, and therefore is expected to produce step sibling or cousin relationships with the original app.

No modification of .dex file:

- *File alignment to 4 and 8 bytes.* zipalign utility [4] is commonly used to optimize the application package, aligning uncompressed data to the specified number of bytes. Although such alignment preserves both the functionality and the content of the internal files, it alters the hash of the .apk container file. Since alignment is a common process in Android app development, for this transformation we employ repackaged apps rather than the originals that are likely to be already aligned. We expect this transformation to produce step sibling of the original app (4-byte alignment) and a twin version (8-byte alignment).

– *Icon change.* Using the aapt tool [1], we replace the original icon.png. The only alteration this transformation introduces is the change of the image file and its timestamp. Thus we consider two versions: a pure image file change (that results in a twin app) and file replacement followed by a repackaging of the original app that produces a step sibling.
– *Junk files' addition.* Throughout our preliminary experiments we noticed the presence of otherwise identical apps with different configuration files. Although not identical, such apps have high similarity. As such using the aapt tool [1], we recreate this scenario by adding to an original app package randomly selected from other unrelated samples configuration or database files. This process does not require repackaging, thus if by chance a file with the same name is present in the original app, it will be updated to a new version with different content. This transformation is expected to result in twin or sibling relations with the original app.

The final set of 792 samples included original 72 apps, their altered versions transformed using the methods described above and their combinations. The resulting apps were randomly signed. This signing process resulted in updated timestamps and author information in meta-data[1].

5.2 Feature Assessment

The selection of the optimal size n has been the subject of many studies. The inherit trade-off in choosing the size of $n$grams is between the accuracy of detection and the size of frequency vector which ultimately affects the performance of the approach. As such, for a given integer n, the number of components of a program's FV is equal to the n^{th} power of the instruction set of the platform or to the n^{th} power of the number of distinct opcodes within the program, if we ignore those opcodes which do not occur in the program.

To determine the optimal size of $n$grams, we experimented with n ranging from 2 to 8. Figure 3 shows the performance of our approach measured as similarity detection accuracy for various parameters. For feature assessment only, similarity detection accuracy was estimated based on n-gram analysis of .dex files, i.e., two apps were labeled as similar if the similarity of their n-grams have exceeded a threshold of 60% (although we experimented with stricter thresholds, this value provided the best overall result).

As the results show, the best performance was achieved with 2-gram opcodes. We also experimented with various amounts of the most frequent n-grams in the range from 100 to 5000 value comparing that performance with the effect of

[1] The dataset can be requested at http://www.iscx.ca/android-data-set

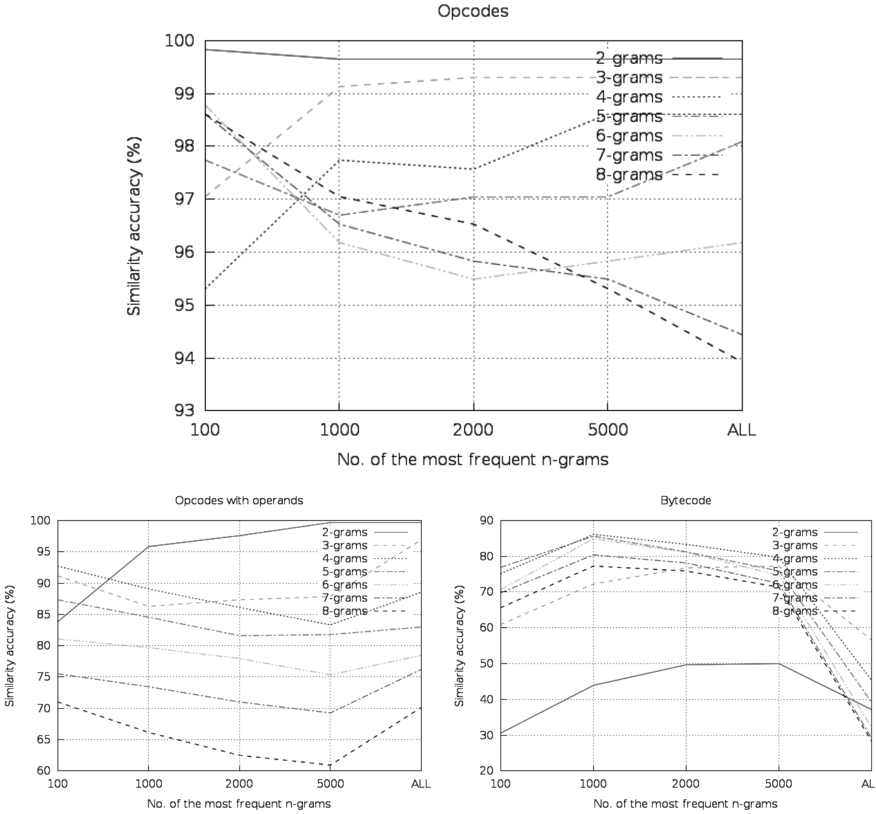

Fig. 3: Similarity accuracy detection for various parameters

retaining all available n-grams. Since 2-gram opcodes' accuracy varied insignificantly (between 99.8% and 99.6%) for all size of frequency vectors, we chose to employ a more memory-efficient variant: 100 most frequent bigrams.

Table 2: The similarity results for Droidkin validation dataset.

(a) Original base set.

Total apps	72	100.00%
No. of unique certificates	58	80.55%
Twins	0	0.00%
Siblings	0	0.00%
False siblings	0	0.00%
Step-siblings	0	0.00%
Cousins	2	2.50%
False step-siblings	0	0.00%

(b) Complete set with transformed apps.

Total apps	792	100.00%
No. of unique certificates	185	23.35%
Twins	168	21.21%
Siblings	167	21.09%
False siblings	46	5.81%
Step-siblings	575	72.70%
Cousins	22	2.78%
False step-siblings	177	22.35%

5.3 Validation Results

The results of experiments with the validation data set are presented in Tables 2a and 2b. As the results show DroidKin mostly confirmed the presence of expected relations in the data. Out of 72 apps, 14 were found to carry the same digital certificate. On the surface this might suggest that all these apps came from the same author. However, upon manual inspection it was revealed that all of them used public keys made available for debugging purposes. Two apps were found to be cousins indicating that they share a large portion of resource files. A closer look showed that apps in these two cases came from the same categories and therefore share approximately 40% of code and employ identical image files.

Among the introduced transformations, 73 apps were found to be unrelated, which means 72 original apps were detected correctly and one app was falsely labeled as unrelated. This false negative appeared from the original app with the smallest code size that was significantly altered with an insertion of junk code. The rest of the apps showed close relations with the corresponding original samples. No false relationships to original samples that did not serve as a basis in transformations were detected.

Table 3: The similarity results.

(a) The Malware Genome dataset.

Total apps	1260	100.00%
No. of unique certificates	134	10.63%
Representative apps	879	69.76%
Unique apps	379	29.61%
Unique apps with unique certificates	86	6.72%
Twins	290	23.02%
Siblings	91	7.22%
False Siblings	2	0.16%
Step-siblings	584	45.63%
Cousins	607	47.42%
False step-siblings	117	9.14%

(b) Drebin dataset.

Total apps	5560	100.00%
No. of unique certificates	963	17.32%
Representative apps	3549	63.83%
Unique apps	1441	25.92%
Unique apps with unique certificates	681	12.25%
Twins	519	9.33%
Siblings	1332	23.96%
False Siblings	136	2.45%
Step-siblings	2365	42.54%
Cousins	2214	39.82%
False step-siblings	1386	24.93%

6 Experimentation

To further evaluate the performance of the proposed approach we employed three datasets made available by the recent studies: The Malware Genome project [36], Drebin [7], DroidAnalytics [33], and we performed a large-scale study of 5,066 Android applications retrieved from Google Play market and 3,116 .apk files retrieved from Virus Total.

The results of similarity analysis are given in Tables 3a, 3b, and 4a. Among the analyzed apps, only 30% are unique apps, i.e., apps that do not exhibit any

Table 4: The similarity results.

(a) Droidanalytics dataset

Total apps	2044	100.00%
No. of unique certificates	232	11.35%
Representative apps	1173	57.39%
Unique apps	773	37.82%
Unique apps with unique certificates	229	11.20%
Twins	545	26.66%
Siblings	211	10.32%
False Siblings	0	0.00%
Step-siblings	960	46.97%
Cousins	703	34.39%
False step-siblings	172	8.41%

(b) Top families in twins/siblings categories (Drebin dataset).

Count	Malware family
Twins	
10	GinMaster
11	Geinimi
13	Adrd
17	FakeInstaller
27	SendPay
41	Kmin
79	DroidKungFu
90	FakeDoc
166	BaseBridge
Siblings	
10	Boxer
20	Kmin
33	Imlog
34	BaseBridge
38	DroidKungFu
402	Opfake
704	FakeInstaller

ties to the rest of the apps. It should be noted that a large portion of them is signed by repetitive keys, indicating that the majority of malware apps come from the same authors.

Among discovered relationships, a significant percent of relatives (30-36%) constitute twins and siblings/false siblings, i.e., apps with identical .dex files. This is an important issue for an evaluation of malware detection methods, as in essence these samples are repetitive and can be recognized with the same set of features. The example of distribution of samples within these categories is shown in Table 4b. Although DroidKin is not designed to detected malware apps, this result shows its ability to reliably group apps based on their content and link them to known malware.

While close relations between samples within one family are expected, cross ties raise many questions. As such, close examination of relations within the Malware Genome data set revealed 197 apps (15.63%) that showed relationships with other families in addition to close ties within the family. Among them only 15 apps (1.19%) had relationships only with other families. Although this might be a result of mislabeling, manual inspection revealed a simple code reuse.

Similarly, in the Drebin dataset, we found 249 (4.47%) apps related to other families as well as their own, and 62 apps (1.11%) with exclusive ties to other families. For example, Anserbot was found to be a cousin of BaseBridge apps (i.e., share the majority of resources), while BaseBridge samples did not exhibit any similarity to the other families.

Based on the analysis of the discovered relationships, we believe that the original sets can be reduced to a smaller set of representative apps that are sufficient to infer the existing similarity among the apps. For example, Drebin data set (5560 apps) can be effectively represented by a set of 3549 apps, which offers a significant reduction and consequently efficiency in analysis.

Using Drebin results as a base, we analyzed a set of apps retrieved from the Google play market and Virus Total. The results are presented in Table 5. While the apps detected in Google market were found to be related to adware, the majority of samples from VirusTotal (155 out of 206) are relatives of LinuxLotoor exploit.

Table 5: DroidKin similarity analysis.

No. of apps	Relations
5066	Google Play market
2	Step siblings
9	False step siblings
	(7 different malware labels)
3166	VirusTotal
206	Relations found
	(18 different malware labels)

6.1 Performance

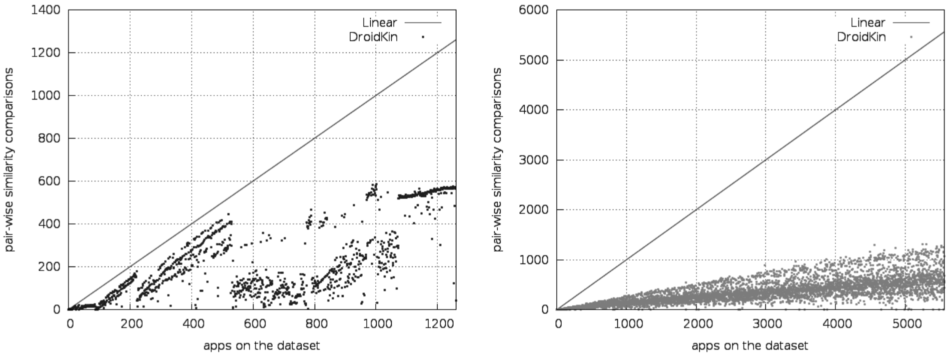

Fig. 4: The run-time performance of DroidKin

The experiments were conducted on an Intel Core i7 @3.40Ghz and 16 GB of RAM. The processing of one application through the initial stage of features extraction on average took only a few milisec. The most 'expensive' parts of analysis are the filtering stage, wich determined a set of candidate apps and similarity scoring of related apps. For example, the complete processing of one unique app (including scoring of 1165 related apps) took 2.57 secs. In total processing of 1280 apps from the Malware Genome data set took 29 minutes, processing the Drebin dataset (5560 apps) took 64 minutes, and DroidAnalytics' data took 10 minutes.

While the performance of DroidKin at these stages highly depends on the amount of feature vectors stored in the database and the uniqueness of a given app, we provide a relative analysis of the number of pair-wise similarity scoring performed on The Malware Genome project' and Drebin data sets. As Figure 4 shows, the largest number of similarity calculations performed for related apps was roughly 600 in Malware Genome data set and 1200 in Drebin dataset, which is considerably less than it would be required for exhaustive pair-wise comparisons.

In the previous experiments, we performed a single thread analysis, but for the large-scale experiment, we employed in parallel seven threads to speed up the computations, resulting in less than 2.5 hours to completely process all apps.

7 Limitations

Although DroidKin is mainly designed to provide an insight for researchers on relations between various apps and guide them through selection and analysis of samples for further study, it showed good detection capabilities. Relying on the analysis of relations between samples, DroidKin effectively groups similar samples without requiring 'expensive' training or predefined detection patterns. This functionality can be enhanced by attributing a group of related apps to specific malware by providing a set of malware samples.

Since DroidKin leverages the application's content, the quality of analysis depends on the size of app. While small code and resource alterations are magnified through the similarity assessment, major changes remain hidden behind a small app size. This is mainly attributed to parasite injections, when benign applications are equipped with a payload in a form of extra classes or files that is in proportion constitute a large chunk of an app's code.

8 Conclusion

With the popularity of the Android platform, the amount of research studies on Android security is rapidly increasing. The value of a study is often dictated by the quality of data employed for the experiments. In this context understanding of relationships behind a diverse set of malware samples becomes an essential step.

In this work we presented DroidKin, a tool for assessing the similarity of Android applications. As opposed to the previous approaches, DroidKin offers deeper insight into app relations, indicating the presence of potential similarity and describing how and why the apps are related. In summary, our experimental results showed:

- *DroidKin is effective* in identifying similarity among apps: as our experiments show DroidKin is able to pinpoint the existing relations correctly introducing a very small misclassification error (1 false positive and 1 false negative).

- Although it is not designed for malware detection, *DroidKin can be potentially leveraged to indicate malicious apps* through the analysis of relatives of known malware samples.
- *DroidKin is efficient*: as opposed to the existing techniques DroidKin can incrementally process apps without training period or predefined detection patterns.
- *DroidKin is robust*: with only 64 min to process 5560 apps DroidKin presents a good alternative for malware detection tools (e.g., Drebin requires 1 day to process 100,000 apps).

9 Acknowledgment

This work was funded by the National Science and Engineering Research Council of Canada (NSERC) through a research grant to Dr. Ali A. Ghorbani.

References

1. aapt tool. http://developer.android.com/tools/building/index.html, May 2014.
2. Virusshare.com - because sharing is caring. http://virusshare.com/, June 2014.
3. Virustotal malware intelligence services. https://www.virustotal.com, June 2014.
4. zipalign tool. http://developer.android.com/tools/help/zipalign.html, May 2014.
5. Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API Level Features for Robust Malware Detection in Android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)*, Sydney, Australia, September 25-27 2013.
6. A. J. Alzahrani, N. Stakhanova, H. Gonzalez, and A. Ghorbani. Characterizing Evaluation Practices of Intrusion Detection Methods for Smartphones. *Journal of Cyber Security and Mobility*, 2014.
7. D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.
8. D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. 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, pages 73–84, New York, NY, USA, 2010. ACM.
9. A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 49–54, New York, NY, USA, 2011. ACM.
10. J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar android applications. In *18th European Symposium on Research in Computer Security (ESORICS)*, Egham, U.K.
11. J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *17th European Symposium on Research in Computer Security (ESORICS)*, volume 7459, pages 37–54. Springer, 2012.

12. M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 23–23, Berkeley, CA, USA, 2011. USENIX Association.
13. N. Eagle and A. (Sandy) Pentland. Reality mining: sensing complex social systems. *Personal and Ubiquitous Computing*, 10(4):255–268, 2006.
14. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.
15. W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.
16. A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, New York, NY, USA, 2011. ACM.
17. J. Forristal. Android: One root to own them all. In *BlackHat*, 2013.
18. G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas. Source code author identification based on n-gram author profiles. In I. Maglogiannis, K. Karpouzis, and M. Bramer, editors, *Artificial Intelligence Applications and Innovations*, volume 204 of *IFIP International Federation for Information Processing*, pages 508–515. Springer US, 2006.
19. C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *11th International Conference on Mobile Systems, Applications and Services (MobiSys)*, Taipei, Taiwan, 2013.
20. M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *The 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 281–294, 2012.
21. S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 62–81, Berlin, Heidelberg, 2013. Springer-Verlag.
22. H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *Proceedings of the 6th International Conference on Trust & Trustworthy Computing (TRUST)*, pages 169–186. Springer Berlin Heidelberg, 2013.
23. M. La Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *Communications Surveys Tutorials, IEEE*, 15, 2013.
24. M. Nauman, S. Khan, and X. Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.
25. S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

26. R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Proceedings of the 4th International Conference on Engineering Secure Software and Systems*, ESSoS'12, pages 106–120, Berlin, Heidelberg, 2012. Springer-Verlag.
27. B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: A perspective combining risks and benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 13–22, New York, NY, USA, 2012. ACM.
28. D. Schreckling, J. Posegga, J. Köstler, and M. Schaff. Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. In *Proceedings of the 6th IFIP WG 11.2 International Conference on Information Security Theory and Practice: Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*, WISTP'12, pages 208–223, Berlin, Heidelberg, 2012. Springer-Verlag.
29. J. Sellwood and J. Crampton. Sleeping android: The danger of dormant permissions. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13, pages 55–66, New York, NY, USA, 2013. ACM.
30. D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of the Seventh Asia Joint Conference on Information Security (Asia JCIS)*, pages 62–69, Aug 2012.
31. L. K. Yan and H. Yin. 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, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.
32. Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 611–622, New York, NY, USA, 2013. ACM.
33. M. Zheng, M. Sun, and J. Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 163–171. IEEE, 2013.
34. W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 185–196, New York, NY, USA, 2013. ACM.
35. W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.
36. Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (SP)*, pages 95–109. IEEE, 2012.
37. Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. 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.