Android authorship attribution through string analysis

Vaibhavi Kalgutkar
University of New Brunswick
Fredericton, Canada
vkalgutk@unb.ca

Natalia Stakhanova
University of New Brunswick
Fredericton, Canada
natalia.stakhanova@unb.ca

Paul Cook
University of New Brunswick
Fredericton, Canada
paul.cook@unb.ca

Alina Matyukhina
University of New Brunswick
Fredericton, Canada
amatyukh@unb.ca

ABSTRACT

With the rising popularity of Android mobile devices, the amount of malicious applications targeting the Android platform has been increasing tremendously. To mitigate the risk of malicious apps, there is a need for an automated system to detect these applications. Current detection techniques rely on the signatures of well-documented malware, and hence may not be able to detect new malware samples. Instead of generating signatures for malware samples themselves, in this work, we propose to develop a lightweight system that can generate signatures of malware writers by leveraging the string components present in their Android binaries. Using these author signatures, we can effectively detect a wide range of existing, as well as any new, malware samples generated by particular authors. The proposed system achieved 98%, 96%, and 71% accuracy over datasets of 1559 benign, 262 malicious, and 96 obfuscated Android applications, respectively. The string-based approach achieved 71% of accuracy compared to only 50% obtained with the existing Ding and Samadzadeh's system.

CCS CONCEPTS

• **Security and privacy → Software and application security**;

KEYWORDS

Android, Authorship attribution, Mobile malware, String analysis, Obfuscation

ACM Reference Format:
Vaibhavi Kalgutkar, Natalia Stakhanova, Paul Cook, and Alina Matyukhina. 2018. Android authorship attribution through string analysis. In *ARES 2018: International Conference on Availability, Reliability and Security, August 27–30, 2018, Hamburg, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3230833.3230849

1 INTRODUCTION

The rapid expansion of Android market attracted an unwanted attention of underground community. The openness of the Android platform and lack of security checks in the application distribution process have resulted in an increasing number of malicious applications targeting the Android platform. An estimated 99% of all existing mobile malware target Android devices [15]. On average anti malware companies see over 8000 novel Android malware samples daily [21].

Based primarily on signatures, i.e., brief and precise rules used to identify malicious behaviour, the anti-malware solutions struggle to keep pace with these unprecedented malware numbers. These signature-based solutions are typically precise, yet unable to spot new malware attacks and require time-consuming human expertise to be assembled for each malware family. To address these gaps, a wide range of malware detection techniques were introduced. The majority of these approaches are based on either the analysis of complex features extracted from the files present in Android executables (e.g., DroidSafe [14]) or dynamic features derived from an app's behaviour during runtime. Extraction of such complex features is a tedious task and requires a considerable amount of time and system resources. With rapid increase in the amount and complexity of malware, it is beneficial to employ methods that do not rely on analysis of complex features and are time efficient.

In this work, we offer an alternative solution. Instead of focusing on individual malware samples, we propose to turn our attention to malware developers. The benefit of this strategy is clear: generating a signature for malware developer rather than individual malware sample, we can effectively characterise all malware samples generated by a particular software developer. The primary premise for our work stems from authorship attribution field, typically used in literary domain. Author attribution domain assumes that each author possesses a unique writing style which can accurately identify his works among others. Our hypothesis is that a developer similar to an author has a unique programming style which is reflected through the various components of programs developed by him. By analysing such program components, we can possibly generate the author's signature that can uniquely identify applications developed by that author. One of the simplest components that can represent an author's writing style is a set of strings used by a developer throughout binary code. Various string components such as variables, class names, method names, and string literals reflect the writing preferences of the developer.

We present an approach to identify an author of an given Android application using the author's signature (profile) comprised of different string components that can potentially reflect the Android author's style. Within our analysis we study different types of strings extracted from Android binary code and Android meta-data. Using these extracted strings, we further generate author profiles by selecting the most frequent word n-grams. N-gram analysis has been used successfully in a variety of research domains including software similarity detection, malware detection, and authorship attribution in the literary as well as the software domain. Due to

ARES 2018, August 27–30, 2018, Hamburg, Germany

ACM ISBN 978-1-4503-6448-5/18/08…$15.00
https://doi.org/10.1145/3230833.3230849

their advantages, we employ the *n*-grams technique to analyse the impact of various strings on an Android authorship attribution.

The authorship attribution in software is associated with many challenges including lack of labelled author data (not to mentioned malware author data). To address this problem we collect Android application both benign and malicious, verify and label them according to their author.

The task of identifying malicious applications is becoming more challenging as code obfuscation in the mobile domain is gaining popularity in both legitimate and malware applications. The goal of obfuscation is to transform original code to disguise its appearance and to protect it from reverse engineering analysis. We also analyze the robustness of our approach in the presence of obfuscation. To this end we create an author dataset with obfuscated Android applications. The proposed system is able to identify the authors of benign, malicious, and obfuscated Android applications with an accuracy of 98%, 96%, and 71% respectively. We also compare our approach to author attribution system proposed earlier by Ding and Samadzadeh [9]. Using the proposed string-based approach, we are able to achieve 71% of accuracy compared to only 50% obtained with the Ding and Samadzadeh's system [9].

2 RELATED WORK

Many of the research studies have provided an overview of mobile malware characteristics, analysis and detection techniques [28]. The existing Android malware analysis studies can be classified into two categories: static analysis based on the features extracted from the application binary [5, 12, 14, 31] and dynamic analysis based on the features derived from an application's runtime behaviour [7, 10, 30, 32].

In contrast to the previous systems based on the analysis of malware samples themselves, we propose a system to detect the malicious Android applications by analysing their authors. Authorship attribution aims to solve such a problem. Authorship attribution in the literary domain has been widely studied since the 19th century. In software domain author attribution is fairly new. Its initial application was in plagiarism detection area where researchers started to examine authors' programming styles based on the analysis of various code components [22].

As the software authorship attribution field evolved, a number of studies started to apply the attribution techniques in other domains including intrusion analysis. Spafford and Weeber proposed to utilize features extracted from code and the remnants of software to identify a potential adversary [27]. They defined this technique as software forensics. Later on, Krsul highlighted the effectiveness of authorship analysis to enhance real-time intrusion detection systems [19]. Traditional software authorship attribution studies were based on the selection of the set of feature metrics to represent the author's style, followed by the classification method to discriminate the authors. The researchers studied a range of features such as number of operators, operands, spaces, keywords, etc [22]. However, most of these proposed metrics were programming language dependent. Moreover, the selection of the metrics was a crucial task. This led to the development of new techniques such as the *n*-gram analysis in the attribution field.

The performance of *n*-grams in the software attribution field was first evaluated by Frantzeskou et al. [11]. Frantzeskou et al. presented a novel system to generate the author profiles by extracting the most frequent byte level *n*-grams from the source code samples. This promising method known as the source code author profiles approach (SCAP) was previously evaluated successfully for literary authorship attribution by Keselj et al. [16]. The system was programming language independent and can be seamlessly applied for the attribution of source code written in any programming language. The system performed surprisingly well even with the limited amount of code samples per author and even in the absence of comments. The study attracted the attention of the research community and underlined the effectiveness of *n*-grams to represent the author's programming style.Function words such as 'the', 'but', 'and', and 'or' have been used extensively in the literary authorship attribution domain as strong markers of an author's style. Based on a similar hypothesis, Burrows and Tahaghoghi presented an efficient system to generate the author fingerprints/profiles by leveraging *n*-grams of function words, i.e., keywords and operators from the source code samples. Later, Kothari et al. evaluated the performance of character based *n*-grams and other stylistic and layout based features for the attribution task [18].

In 2014, Burrows et al. compared the different source code authorship attribution techniques [8]. The study confirmed the effectiveness of *n*-gram analysis as presented in the SCAP method [11] for the source code attribution task. The *n*-gram analysis technique proved to be effective for similar research problems as well. Layton et al. extended the byte level *n*-grams to analyse phishing websites with an authorship perspective [20]. In another study, researchers evaluated the performance of byte *n*-grams for computer virus detection [26]. As the field of authorship attribution evolved, researchers studied a variety of string-based features such as keywords, operators, operands, variables and function names to quantify authors' styles. In this work, we further investigate the impact of strings extracted from Android binary code and meta-data on authorship attribution.

3 STRING-BASED ANALYSIS

An Android application package file (APK) contains a variety of files including an executable file also known as a *DEX* file. It contains compiled Android classes in *.dex format* for Android Runtime. The DEX file is partitioned into a number of sections such as a header section followed by several identifier lists such as string, type descriptors, prototype, field, method, class definitions and a data section containing the actual implementation data. Every element defined in the identifier list sections maintains the offset pointing to the corresponding entries in the data section. For example, the string identifier list section maintains offsets of all types of strings used in the DEX file. The string offset points to the location in the data section where the string has been used. Other identifiers sections such as type ids, prototype ids, field ids, method ids, and class defs, also maintain the list of offsets pointing to the data section where the identifiers have been actually used. However, other than maintaining references to the data section, these sections also contain references to the string identifier list. For example, a class named *AccountActivity* will have a reference to

Table 1: Summary of different types of strings analysed

String type	Description
Unreferenced	Strings referenced only by the string identifier list of the DEX file
DEX	All the strings components present in the DEX file
Application	Strings extracted from the *strings.xml* file
All	All the types of strings combined together

the class data in the data section as well as a reference to the actual string *AccountActivity* defined in the string identifier section.

Although Android application development should follow the specifications and guidelines stated by the official Android Development community, malware writers often violate these specifications to include malicious code inside the Android application. One of the approaches is to *hide* the malicious code by placing it in the data section of the DEX file while avoiding it being referenced by elements of the identifiers sections such as class ids or method ids [2, 3]. The identifier lists in the DEX files are typically used to invoke the various methods and classes and are useful for analysing the workflow of the application. However, the *code hiding* technique prevents the reverse engineering of an application, making it difficult to recognize malicious functionality.

Strings used in such hidden code do not have references to identifier sections other than the string ids list. Thus, we differentiate the strings present in the data section of the DEX file as the *'referenced strings'* and the *'unreferenced strings'*. The referenced strings have a reference to the identifier list other than the string offset list, whereas the unreferenced strings are referred to only by the string identifier list. The referenced strings represent classes, methods, or different data types in the application depending on the type of the identifier list referring to them. Thus, these strings form a part of the functional, executable code of the application. On the other hand, the **Unreferenced strings** are only referenced by the string offset list. These strings can contain hidden, interesting or malicious code details. For example, malware writers often use hard coded URLs, email addresses, and malicious code samples in the form of strings in malware applications. Thus, analysis of such hidden unreferenced strings can reveal malicious activity embedded in Android applications. We therefore analyse unreferenced strings found within the DEX file of the Android application. We also analyse the impact of all the **DEX strings** components, i.e., by combining both unreferenced and referenced strings for the authorship attribution task.

Other than the DEX strings present in the DEX file, an APK contains another source of string components, i.e., the XML resource file *'strings.xml'*. Theoretically, it should provide a list of all the strings that can be referenced within the application source code or from other resource files of the application [25]. However, Android developers may choose what strings to define there and even whether or not to define any at all. As a result *'strings.xml'* might be a curated version of what a developer wants users to know about his application.

These author-defined strings can be a strong style marker denoting the author's writing style and pattern. We refer to strings present in *strings.xml* file as the **Application strings**. We examine the author style based on these strings as well.

Thus, we evaluate the performance of *unreferenced*, *DEX*, *application*, and **All strings** (i.e., by considering all types of strings together) for the task of identifying the author of Android applications. Table 1 summarises types of strings used for the analysis.

4 AUTHOR ATTRIBUTION SYSTEM

The overall flow of the proposed system is depicted in Figure 1. Given an Android application to be attributed to an author profile, our system first extracts different types of strings from the Android APK, then generates string *n*-grams, and then forms feature vectors. This feature vector is then compared to existing author profiles (generated earlier) to predict a potential author. Each of this steps is explained in more details in the following sections.

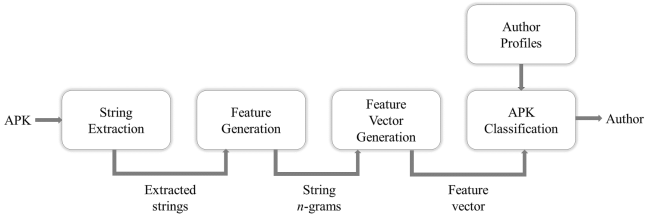

Figure 1: The flow of the proposed system

4.0.1 String Extraction. The attribution system first unpacks every APK under the analysis. To analyse the *DEX strings*, the system first extracts all the *classes.dex* files from the unpacked APK, and then traverse the string ids list to extract all the strings referenced by the list. However, to analyse only the *Unreferenced strings* present in data section of the DEX file, the system further discards the *referenced strings* from the set of *DEX strings* by traversing the other identifier lists, i.e., type, prototype, field, method, and class lists. For the extraction of *Application strings*, the system extracts *strings.xml* from the unpacked APK. The system then parses this *strings.xml* file to extract the strings defined by the attribute *'name'* of the XML *'string'* elements. In the case of *All strings* analysis, all types of strings contribute to the final string data. All the extracted strings form the basis for the string *n*-grams analysis.

4.0.2 Feature generation. The system employs word level string *n*-grams derived from the extracted strings as features for the author classification task using sliding window techniques. Word level *n*-grams can be viewed as a sequence of *n* words combined together from a given sentence or text. During the feature generation step, the system generates the list of extracted strings in lexicographical order and outputs each extracted string on a separate line. The word level *n*-grams are generated by combining the words from a string on every line. This is done in order to generate line bounded *n*-grams. The advantage of line bounded *n*-gram is that it does not contain words from different lines. As strings on different lines

Table 2: String *n*-grams feature examples

1-grams	2-grams	3-grams	4-grams
mSet=	<LB>mSet= mSet=<LB>	<LB>mSet=<LB>	
Value must not be null	<LB>Value Value must must not not be be null null<LB>	<LB>Value must Value must not must not be not be null be null<LB>	<LB>Value must not Value must not be must not be null not be null<LB>

are not necessarily associated, combining the words belonging to different lines would have resulted in the generation of *n*-grams containing words that are not a part of a single string, therefore generating *n*-grams exhibiting meaningless semantics. While generating line bounded word *n*-grams, the system ignores a line if it represents a string with less than *n* words. This can cause a loss of the information. Thus, the system augments each extracted string with a unique line boundary token at the start and end of the string before extracting *n*-grams. The addition of line boundary tokens ensures that every line contains at least 3 words including the line boundary tokens. Thus, more features are available to represent an author's style.

The line boundary tokens also give information about the context of the first and the last word. This can provide crucial information for example, a string beginning with words like *Error* or *Warning* can mean something different than a string containing such words elsewhere. Line boundary tokens are not added for generating *uni*-grams (1-grams) as *uni*-grams represent only a single word.

Table 2 demonstrates the *n*-gram features generated from two strings — *mSet=* and *Value must not be null*. From the table it is clear that there are no line boundary tokens (<LB>) added for the 1-gram features. Also, in the case of a string containing only a single word (e.g., *mSet=*), there is no 4-gram feature available as even with line boundary tokens the given string contains only 3 words.
The generated word level *n*-gram features are further used to form feature vectors.

4.0.3 Feature vector generation. The system produces the numerical representation of the generated *n*-gram features in the form of a *vector* suitable for machine learning classification. In the feature vector generation process, the system calculates frequencies of extracted word *n*-grams to represent each APK under analysis in the form of a *frequency vector*. Every element in the *feature vector* represents the number of times a particular *n*-gram occurs in a given APK. For example, consider the following strings extracted from an APK:

- Cookie value must not be null
- Cookie name must not be empty

This APK will results in 11 unique word level 3-gram features, i.e., [<LB> Cookie value, Cookie value must, value must not, must not be, not be null, be null <LB>, <LB> Cookie name, Cookie name must, name must not, not be empty, be empty <LB>] which can be represented in the form of *frequency vector* as:

$$v = [1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1]$$

Here, each element in the vector represents the frequency of the corresponding *n*-gram feature. The system generates feature vectors for every Android application of each author. However, different applications can have different *n*-gram features which can lead to feature vectors having different sizes. We transform frequency vectors into a common and consistent feature space using count vectorization technique that creates vectors with all unique sequence of *n*-gram features extracted from all the applications in the dataset and their corresponding frequency. In this case the generated feature vectors are sparse, i.e., contain elements with zero frequencies. To remove inefficiency, we employ sparse matrix approach implemented by Python skit-learn library that only retains non-zero occurrences.

4.0.4 APK classification. Our Android attribution system employs feature vectors generated in the previous stage as input data for the supervised classifier. To generate a profile for a known author several APKs, i.e., several feature vectors representing these APKs, are analyzed by our system. The system trains the supervised machine learning classifier in order to produce the unified profiles for each of the authors in the dataset. Using these *inferred* author profiles, this trained classifier model can then predict the authorship of any unknown Android application.

We selected support vector machine (*SVM*) as the supervised classifier for our system. The support vector machine classifier is one of the robust classifiers [1] and has successfully been applied in attribution domain [6]. It has also been successfully used for Android malware family classification and gave the best performance among a wide range of machine learning classifiers [17]. Thus, here we present results with SVM classifier.

5 DATA

The availability of a dataset (i.e., a labelled set of Android developers with their Android application samples) is a major challenge for any authorship attribution study. There are no standard open benchmark datasets available in this domain. Thus, we created three new datasets of Android samples belonging to different authors — benign authors dataset, malware authors dataset, and obfuscated application dataset — which we use for evaluation.

5.0.1 Benign application dataset. According to the specification stated by the official Android community, each Android APK developer needs to digitally sign the APK with a certificate which contains the information that uniquely identifies the developer of the APK, such as the public key. The developer possesses the corresponding private key for the public-key certificate. [24]. We used the certificates attached to the Android application to collect the samples belonging to the same author. APKs having the same certificates are considered to be coming from the same author/developer. While Android developers can generate different certificates to sign each new APK, we do not consider this case and follow a more conservative approach to ensure that we have ground truth data.

To build the dataset of legitimate authors, we collected more than ten thousand Android APKs from eight different Android markets (the official Android market — Google play store, Appland, Anzhi, Aptoide, MoboMarket, Nduoa, Tencent, Xiaomi). We grouped these APKs by their certificates. Thus, APKs belonging

to the same author/developer were grouped together. Some certificates are publicly available and anyone can use them for signing the APK. Keeping this in mind, we discarded the APKs signed by such public certificates. We also discarded APKs signed by debug certificates. We removed duplicate APKs that have been published in multiple markets. Finally, for our dataset, we considered only authors having more than 20 applications. The final dataset contains a total of 40 authors with 1559 Android applications.

5.0.2 Malware application dataset. As the authorship information of many malware samples is not known, the task of collecting the malware samples grouped by authors is difficult. However, there are Android malware research communities that provide a platform to share malicious Android APKs. We built the malware authors dataset using *'koodous'* system which is an open source, collaborative, web based Android malware analysis platform[1]. We used the *search API* provided by the system to search for malicious APKs with the same certificate. We collected a total of 262 malicious APKs from 10 different authors with each author having at least 10 unique malicious APKs. The malicious nature of APKs was also verified by VirusTotal platform.

5.0.3 Obfuscated application dataset. Due to a wide range of advantages, Android obfuscation is gaining popularity among both legitimate as well as malware writers. As obfuscation affects the underlying APK, it is important to test its impact on our analysis. However, collecting obfuscated Android APKs is a challenging task.

There are research studies focusing on the detection of the type of obfuscation used in Android binaries [4, 29]. However, none of the proposed methods is 100% accurate and therefore, we can not rely on these techniques to identify and collect obfuscated APKs. We choose to build an obfuscated dataset using Android application source code. Generating obfuscated APKs from the source code is beneficial as most of the Android obfuscation tools work at the source code level.

The open source community *'GitHub'* is one of the largest collaborative platforms which provides the ability to host and share multiple projects through repositories[2]. We collected the source code of Android projects hosted by different Android authors from GitHub. We used the API provided by GitHub to search for the Android repositories. We then employed different obfuscation tools on these source code projects to generate the dataset of obfuscated applications. As GitHub is a collaborative platform, a single repository can have multiple contributions. Thus, to ensure that a repository doe not contain code belonging to multiple authors we discarded the forked repositories and repositories having more than one contributor. We also examined each repository manually to ensure that it does not contain code to include external repositories during the application compilation process.

We initially collected more than 255 Android projects from 28 authors. However, many of the projects could not be compiled due to various reasons e.g., being unable to locate external dependencies files, compilation errors due to incomplete code or improper project settings. We discarded authors having very few Android projects as enough samples are needed to represent the author's style. We were finally able to collect a total of 96 source code projects belonging to 9 authors. These apps were then obfuscated using 3 popular Android obfuscation tools — *ProGuard*, *Allatori* and *DashO*. These tools are considered as a Java class file obfuscator as they obfuscate Java class files of the Android application which are compiled further to produce DEX files. ProGuard performs three major steps for obfuscating the Android APK. Initially, it performs code shrinking by removing unused code such as unused methods, variables, and classes followed by Java bytecode optimization. Finally, it performs name obfuscation, i.e., it obfuscates the code by renaming variables, classes, methods, and fields to some short random meaningless words.

Table 3: Obfuscation tools and techniques used for the analysis

Obfuscation tool	Obfuscation Techniques
ProGuard	Code shrinking String obfuscation
Allatori	String obfuscation String encryption Control flow obfuscation Optimization
DashO	String obfuscation String encryption Control flow obfuscation Optimization

We used the default ProGuard configuration file without any optimization to generate the obfuscated APKs as it is sufficient for removing unused code and for obfuscating Android applications. Like ProGuard, Allatori and DashO have the provision of string obfuscation. However,they provide features such as string encryption, control flow obfuscation, and optimization in addition to name obfuscation by renaming variables. Table 3 summarises the tools used for the obfuscation. We used the default configuration setting provided by these tools which has all of these mentioned features to generate the obfuscated APKs. We used these tools to obfuscate the 96 Android source code projects cloned from GitHub. Thus, each Android application project produced one unobfuscated APK and 3 versions of the obfuscated APKs using the above obfuscation tools. Table 4 summarises the datasets used for the analysis.

Table 4: Summary of datasets

Dataset	Authors	Apps	Description
Benign	40	1559	Benign applications collected from 8 different Android markets
Malicious	10	262	Malicious applications collected from *Koodous* system
Obfuscated	9	96	Collected form *GitHub* platform. Each project produced 1 unobfuscated and 3 obfuscated APKs using different obfuscation tools

[1]https://koodous.com/
[2]https://github.com/

6 EXPERIMENTAL RESULTS AND DISCUSSION

6.1 Implementation details

We implemented the proposed system in the Python language. We used various Python modules available in the **scikit-learn** library [23]. Another important tool used in our analysis is **DroidKin**. DroidKin is a lightweight tool used to measure the similarity between Android applications [13]. It provides similarity scores between APKs which can be used to detect DEX files having identical or similar content. We used this similarity score as a threshold value. We evaluated the system performance over the dataset generated after discarding all the APKs having equal or greater similarity score than a specified threshold. Discarding the similar APKs will lead to reduction of more frequent string *n*-gram features. Thus, this analysis can give us the idea of the robustness of our system over datasets having varied levels of APK similarity.

For evaluation, we adopted a 5-times 5-fold stratified cross validation strategy. To ensure the same set of cross validation folds were used for experiments with different features (i.e., string types), we used a random seed. We calculated accuracy, macroaveraged precision, macroaveraged recall and macroaveraged F_1 scores for evaluating the system performance.

6.2 Preliminary analysis

We conducted a variety of preliminary experiments to examine different parameters for our system. Initially, we analysed the system performance by changing the size of word *n*-grams. We observed the improvement in system performance with increase in size of *n*-grams from 1 to 3. However, the system performance degrades with further increase in size of *n*-grams. Our system ignores the strings having less than *n* terms even after addition of line boundary tokens. Furthermore, many 4-grams or even higher order *n*-grams will be unique leading to data sparsity. Thus, we used 3-grams for further experiments.

We compared the performance of plain frequency count feature vectors with that of *tf-idf* (term frequency and inverse document frequency) count feature vectors. Tf-idf vectors did not enhance the system performance; hence, we employed the frequency count feature vectors for the rest of the experiments.

Traditional malware detection systems analyse features at the bytecode or opcode level. These are useful for representing the application at a low level. Thus, we compared the performance of our proposed attribution system based on string *n*-gram features with bytecode and opcode *n*-grams features. We evaluated performance of bytecode *n*-grams with *n* ranging from 1 to 2 and opcode *n*-grams with *n* ranging from 1 to 4. The best performance was observed for 2-gram bytecode features and 3-gram opcode features. Thus, we used these representations for further experiments.

6.3 Benign application dataset results

6.3.1 Comparison of different types of strings. We compared the performance of our system over four types of strings — *Unreferenced strings* — strings referenced only by the string identifier list of the DEX file, *DEX strings* — all the strings components present in the DEX file, *Application strings* — strings extracted from the *strings.xml* file, *All strings* — all the types of strings combined together. We also compared the performance of the system using bytecode and opcode. Table 5 illustrates the performance of the Android attribution system over the benign application dataset.

Table 5: Performance comparison of different types of features over the benign application dataset

Type	Accuracy	Macro Average F1	Average Training Time (Seconds)
All strings	98.19%	97.55%	606.78
DEX strings	98.17%	97.55%	625.30
Unreferenced strings	97.55%	96.64%	208.81
Application strings	94.40%	93.10%	5.53
Bytecode	32.15%	28.38%	135.72
Opcode	87.83%	86.92%	272.79

As expected All and DEX strings produce the best results as they preserve the majority of strings found in an APK. Nevertheless, the training time of the classifier is significantly reduced in the case of *unreferenced* and is lowest for *application* strings. A total of around 36 million strings are processed in the case of *All* and *DEX* strings which is reduced drastically in the case of *unreferenced* and *application* strings (to around 12 million and 380 thousand strings, respectively). The system performs quite well even with the lower number of string components. Results also demonstrated that string based features are better than bytecode and opcode features.

6.3.2 Effect of APK similarity. For our datasets we only collected Android APKs with unique MD5 hash value. While this is a common practice to remove redundant samples, this is not sufficient as authors commonly repackage their applications adding/modifying functionality of the code. To understand the impact of code similarity on system accuracy, we employed the DroidKin tool that provides a pairwise similarity score between APKs.

Using this similarity score as a threshold value, we analyse out data. All the APKs of an author having a similarity score equal or greater than the specified thresholds are discarded from the dataset as being too similar to each other. After removing similar APKs, authors having less than 5 APKs are also discarded, as at least 5 APKs are needed to perform stratified 5-fold cross validation experiments and have at least one app from each author in each fold. The performance of the system over the benign application dataset at varying similarity thresholds is illustrated in Figure 2.

The results are quite predictable. The system performance gradually degrades as APK similarity level threshold decreases. Removal of similar APKs from an author leads to a reduction of frequently appearing strings, which affects the frequency count of string *n*-gram features used to represent the author's style. Another contributing factor is the decrease in the size of the dataset with every similarity threshold. Initially, the dataset contains 40 authors with 1559 APKs, whereas, the dataset with a similarity threshold of 65 contains 20 authors with only 270 APKs. Fewer features are available to represent an author's style with a lower number of APKs per author. Thus, the performance of the attribution system is lower on a smaller dataset. Nonetheless, our system performance is quite promising

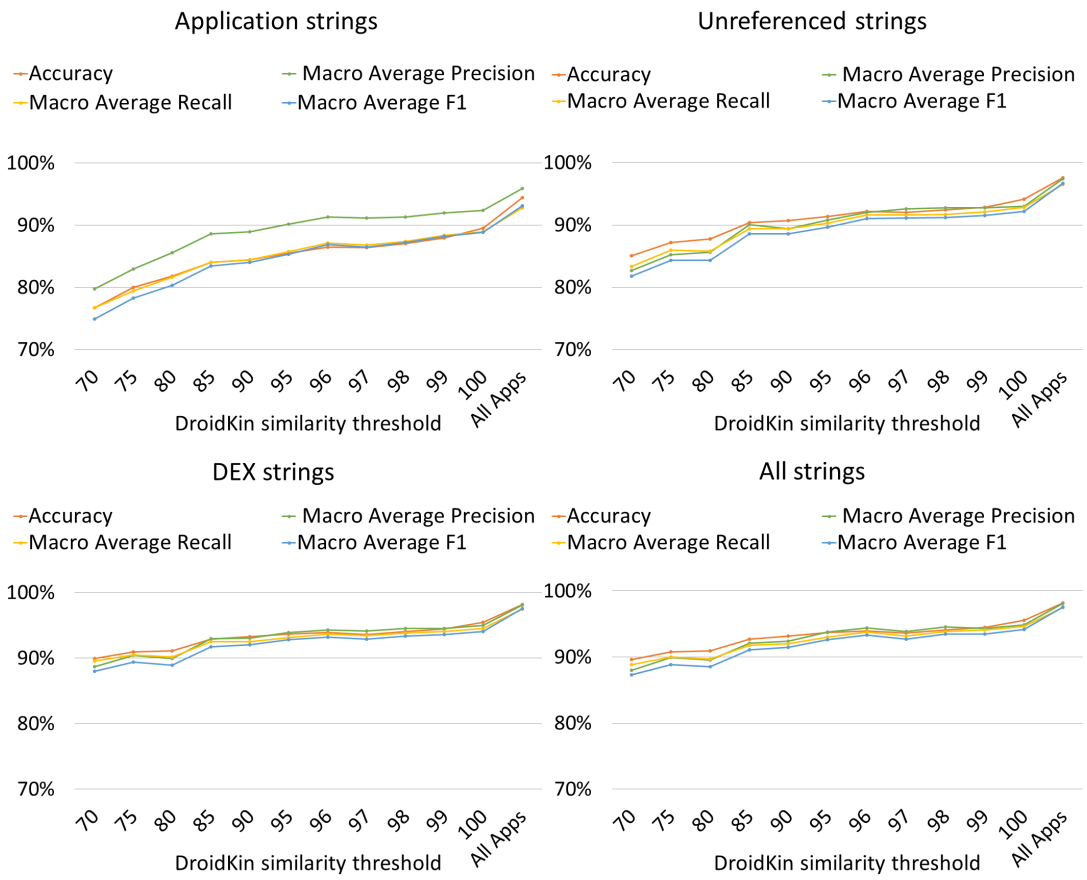

Figure 2: APK similarity threshold vs system performance on the benign dataset

even with the small amount of data and lower number of APKs. For example, even at the similarity threshold of 65%, the system maintains an accuracy of around 88% for *All* and *DEX* strings; whereas *unreferenced* and *application* strings are able to predict the author of an unseen APK with an accuracy of 84% and 78%, respectively.

6.4 Malware application dataset results

6.4.1 Comparison of different types of strings. We compared the performance of our system over various types of strings, bytecode and opcodes extracted form malicious applications. These results are shown in table 6. Similar to the performance of different types of strings over the benign application dataset, in the case of the malware dataset, the *All* strings approach performs better than all other types of strings due to the higher number of strings retained. However, the *unreferenced* and *application* strings provide satisfactory results even with a smaller amount of strings and lower training time. Although application strings show comparable results, this string set is the least reliable. Application strings are derived form string.xml file that is fully dependent on a developer intentions and thus can be carefully curated to only include selected strings. From this perspective DEX and unreferenced strings are able to provide more trustworthy author profile. These are the strings that are embedded within the code and often unintentionally selected by a developer.

Table 6: Performance comparison of diffierent types of features over the malware application dataset

Type	Accuracy	Macro Average F1	Average Training Time (Seconds)
All strings	96.03%	95.99%	20.80
DEX strings	95.78%	95.78%	21.44
Unreferenced strings	94.76%	94.70%	8.09
Application strings	81.63%	82.12%	0.33
Bytecode	37.76%	39.32%	9.36
Opcode	89.20%	89.13%	29.98

The system shows better results over the benign application dataset (discussed in the previous section) than the malware application dataset. This is because the malware application dataset is smaller in size than the benign application dataset. Thus, fewer samples are available for analysis of a given malware author's style. However, even with limited samples, our system produces good accuracy and demonstrates that it can effectively identify malware

writers as well. Bytecode based features perform poorly. Though, opcodes results are better than bytecode features, it could not outperform *all*, *DEX* and *unreferenced* strings approach.

6.4.2 Effect of APK similarity. The results of these experiments are shown in Figure 3. Results on the malware dataset follow similar trends to those on the benign application dataset; i.e., performance of the system degrades with the similarity threshold. However, the performance is quite volatile due to the small size of the dataset which is further reduced for the lower values of similarity threshold. For example, initially the dataset contains 10 authors with 262 APKs. Even at a similarity threshold of 100% (in this case only APKs having identical DEX files are removed), 81 APKs are removed from the original dataset leaving only 181 samples for the analysis. With every subsequent similarity threshold value, the number of APKs in the dataset is reduced, making the dataset smaller. Thus, we observe a few spikes in the performance charts shown in Figure 3. Nevertheless, even with a small number of samples, our system can still attribute malware authors. For example, at the 65% similarity threshold with only 55 APKs and 5 authors available in the dataset, the system was able to achieve accuracies of around 86% using *All* and *DEX* strings, around 84% for *unreferenced* strings, and 77% for *application* strings.

6.5 Obfuscated application dataset results

We compared the performance of our system over all 3 versions of the obfuscated dataset, as well as the unobfuscated version of this dataset. We again compared performance of different types of strings, bytecodes and opcodes. The results are presented in Table 7. As shown in the table, without obfuscation, our system is able to predict the author of an Android binary with an accuracy of almost 71% for *All* and *DEX* strings, 70% for *unreferenced* strings, and 62% for *Application* strings.

The experiments performed until now highlighted the correlation between the size of the dataset and system performance. From the given dataset of only 96 APKs, the system was able to extract a total of around 1.4 million *all* strings (this number was around 36 million for the benign dataset and 5.4 million for the malware dataset). Whereas, in the case of *unreferenced* strings, this number dropped to 400k, and for *application* strings it was only 2689 strings. The smaller number of strings is a likely reason why the system was not able achieve accuracy as high as the benign or malware application datasets. Another factor we must consider is that all the source code Android projects are downloaded from the GitHub repository, which is an open source collaborative platform. Even with all the verifications checks and precautions (discussed in the section 5.0.3), there is a possibility that a program is written by multiple authors or contains code components directly copied from other sources. We also ignored many of the source code projects for several reasons such as compilation errors, incomplete code and missing library files. Such excluded projects might contain important information about an author's writing style but were not considered in our analysis. All these factors could have led to the generation of noisy author profiles. Yet, the results are quite promising, and the system was able to predict authors with an accuracy of 71%.

In the case of the obfuscated dataset experiments, except for the *application* strings, the system performance degrades slightly in the case of Allatori and moderately in the case of the DashO obfuscation tool. Obfuscation changes the names of various string components such as classes, methods, and fields to random strings. Along with the obfuscation, Allatori and DashO both have the provision of string encryption. Encryption encodes a given string to produce a new unreadable string. In both cases, the original string is converted to a random string. Such random strings are not informative for the authorship task and can produce noisy author profiles. Nevertheless, system performance is not affected significantly for the Allatori and Dasho obfuscated datasets, even after obfuscation and encryption of string components.

The performance of the ProGuard obfuscation tool is quite interesting. In the case of *unreferenced* strings, the system performance is affected slightly as compared to the unobfuscated dataset. This is not surprising as ProGuard obfuscates various string components and this affects the classification performance. However, in the case of *All* and *DEX* strings, the ProGuard dataset results are better than for the unobfuscated dataset. The way in which the ProGuard obfuscation operates may be the reason of such an unusual performance. With the default obfuscation settings, ProGuard extensively removes various unused code components such as debug information and unused code. This is reflected in the number of strings available for analysis in the ProGuard obfuscated dataset. For example, the number of *All* or *DEX* strings extracted in the case of all other datasets except ProGuard (unobfuscated, Allatori, and DashO) is around 1.4 million. In the case of the ProGuard obfuscated dataset, this number reduced drastically to only 377k strings. This might have led to removal of noisy data from the samples, and therefore focusing on only relevant features representing an author's style. Unlike *unreferenced* strings which mostly contain only user-defined code components, *All* and *DEX* strings contain library and third party classes as well. Hence, removal of redundant information could have eliminated strings that are not very informative for authorship analysis. Also, the size of the datasets is rather small and thus a few different classification predictions can influence accuracy.

As none of the obfuscation tools used for our analysis obfuscate *application* strings, the system has the same performance for this type of string over all of the obfuscated and the unobfuscated datasets.

We observed a similar trend for opcodes and bytecodes as for the benign and malware application dataset. The string based approach outperforms opcode and bytecode based features.

To study the performance of the system over the obfuscated datasets with varied levels of APK similarity, APKs having higher similarity score than the given similarity threshold must be removed from the dataset. Thus, the number of APKs in the dataset is reduced with the similarity threshold levels. As the number of APKs is already relatively small (only 96 APKs), this would lead to a very small dataset, and the results over such a small dataset would not be very informative. Hence, we did not include this analysis in our work.

We also compared the performance of our system with the Java attribution system proposed by Ding and Samadzadeh [9]. This system extracts 56 features based on layout, style and structure of the program for attributing Java source code files. We evaluated this

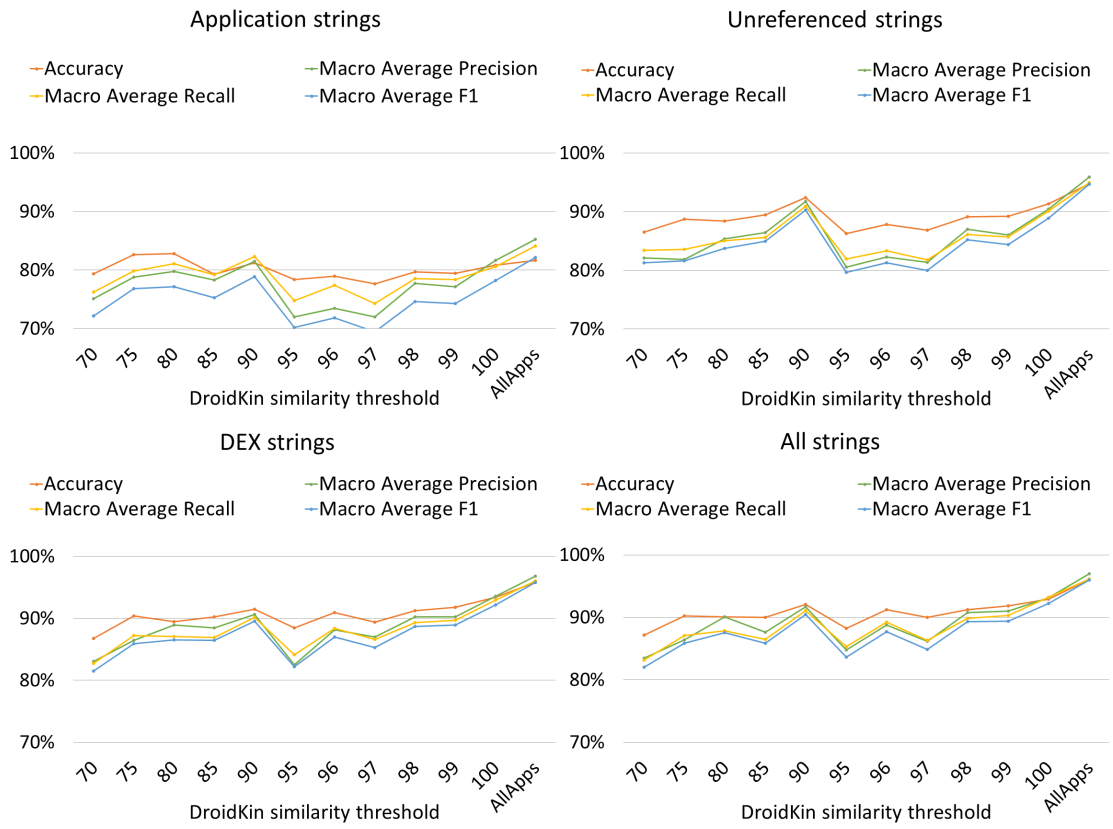

Figure 3: APK similarity threshold vs system performance on the malware dataset

Java attribution system over the Java source code files present in 96 unobfuscated Android applications. The system was able to predict authors with an accuracy of only 49.68% which is significantly less than the 71% accuracy achieved with our proposed attribution system.

7 CONCLUSION

In this work, we presented a lightweight, efficient system to identify the author of an Android binary by analysing string components present in the application. Our results showed that we can predict the author of an Android binary just by analysing string components at the binary level.

To the best of our knowledge, this is the first attempt to design an Android authorship attribution system leveraging string data within APKs. The system proved to be very efficient as it can handle many android samples with a relatively low classification training time. We compared the performance of different types of string components such as *all*, *DEX*, *unreferenced* and *application* strings for the task of Android authorship attribution. In terms of system accuracy, *All* and *DEX* strings outperform *unreferenced* and *application* strings. However, *unreferenced* and *application* strings substantially reduce the number of strings analysed and overall time required for training the system, while maintaining fairly good accuracy.

We compared the performance of opcode and bytecode features with our proposed string approach. Our string based approach performs better than these techniques. The proposed approach also outperformed the existing Ding and Samadzadeh's Java attribution system.

Another important contribution of our work is the datasets used for our analysis. In the authorship attribution domain, the lack of open benchmark datasets is a serious challenge faced by researchers. We generated 3 different datasets to study benign, malware and obfuscated Android applications. These datasets will be helpful for further research in Android authorship attribution.

REFERENCES

[1] Mohamed Aly. 2005. Survey on Multiclass Classification Methods.
[2] A. Apvrille. 2012. Guns and smoke to defeat mobile malware. In Hashdays Conference..
[3] A. Apvrille. 2013. Playing hide and seek with dalvik executables. In Hacktivity.
[4] Axelle Apvrille and Ruchna Nigam. 2014. Obfuscation in android malware, and how to fight back. *Virus Bulletin* (2014), 1–10.
[5] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *NDSS*.
[6] Christopher JC Burges. 1998. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery* 2, 2 (1998), 121–167.
[7] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 15–26.

Table 7: Performance comparison of different types of features over the datasets obfuscated with different tools

Obfuscation Type	Type	Accuracy	Macro Average F1	Ave. Training Time (Seconds)
No obfuscation	All strings	70.68%	64.79%	14.38
	DEX strings	70.57%	64.36%	14.26
	Unreferenced strings	69.68%	64.37%	4.95
	Application strings	62.20%	53.46%	0.02
	Bytecode	28.98%	21.51%	2.15
	Opcode	61.69%	56.03%	30.79
ProGuard	All strings	76.96%	72.07%	3.68
	DEX strings	76.91%	71.94%	3.65
	Unreferenced strings	65.52%	59.19%	1.03
	Application strings	62.20%	53.46%	0.02
	Bytecode	18.57%	11.10%	1.56
	Opcode	70.40%	64.41%	16.32
Allatori	All strings	70.00%	63.76%	12.97
	DEX strings	69.23%	62.69%	13.02
	Unreferenced strings	66.15%	59.84%	4.42
	Application strings	62.20%	53.46%	0.02
	Bytecode	23.38%	16.40%	2.05
	Opcode	62.25%	56.02%	28.85
DashO	All strings	67.44%	60.30%	14.68
	DEX strings	67.42%	59.90%	14.66
	Unreferenced strings	64.26%	56.46%	5.08
	Application strings	62.20%	53.46%	0.02
	Bytecode	19.83%	13.45%	22.76
	Opcode	60.33%	53.18%	28.22

[8] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin. 2014. Comparing techniques for authorship attribution of source code. *Software: Practice and Experience* 44, 1 (2014), 1–32.

[9] Haibiao Ding and Mansur H Samadzadeh. 2004. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software* 72, 1 (2004), 49–57.

[10] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[11] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, Carole E Chaski, and Blake Stephen Howald. 2007. Identifying authorship by byte-level n-grams: The source code author profile (scap) method. *International Journal of Digital Evidence* 6, 1 (2007), 1–18.

[12] Yanick Fratantonio, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2015. Clapp: Characterizing loops in android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 687–697.

[13] Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. 2014. Droidkin: Lightweight detection of android apps similarity. In *International Conference on Security and Privacy in Communication Systems*. Springer, 436–453.

[14] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe.. In *NDSS*.

[15] AV TEST The Independent IT-Security Institute. 2016. *The AV-TEST Security Report 2015/2016*. Technical Report.

[16] Vlado Kešelj, Fuchun Peng, Nick Cercone, and Calvin Thomas. 2003. N-gram-based author profiles for authorship attribution. In *Proceedings of the conference pacific association for computational linguistics, PACLING*, Vol. 3. 255–264.

[17] Stakhanova N Killam R, Cook P. 2016. Android Malware Classification through Analysis of String Literals. In *First Workshop on Text Analytics for Cybersecurity and Online Safety (TA-COS 2016)*. European Language Resources Association (ELRA).

[18] Jay Kothari, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis. 2007. A probabilistic approach to source code authorship identification. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*. IEEE, 243–248.

[19] Ivan Krsul and Eugene H Spafford. 1997. Authorship analysis: Identifying the author of a program. *Computers & Security* 16, 3 (1997), 233–257.

[20] Robert Layton, Paul Watters, and Richard Dazeley. 2010. Automatically determining phishing campaigns using the uscap methodology. In *eCrime Researchers Summit (eCrime), 2010*. IEEE, 1–8.

[21] Christian Lueg. accessed June 16, 2017. 8,400 new Android malware samples every day. https://www.gdatasoftware.com.

[22] Paul W Oman and Curtis R Cook. 1989. Programming style authorship analysis. In *Proceedings of the 17th conference on ACM Annual Computer Science Conference*. ACM, 320–326.

[23] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

[24] The Android Open Source Project. accessed July 13, 2017. Publish Your App. https://developer.android.com/studio/publish/index.html.

[25] The Android Open Source Project. accessed July 13, 2017. String Resources. https://developer.android.com/guide/topics/resources/string-resource.html.

[26] D Krishna Sandeep Reddy and Arun K Pujari. 2006. N-gram analysis for computer virus detection. *Journal in Computer Virology* 2, 3 (2006), 231–239.

[27] Eugene H Spafford and Stephen A Weeber. 1993. Software forensics: Can we track code to its authors? *Computers & Security* 12, 6 (1993), 585–595.

[28] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 76.

[29] Yan Wang and Atanas Rountev. 2017. Who changed you?: obfuscator identification for Android. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 154–164.

[30] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. ProfileDroid: multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*. ACM, 137–148.

[31] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 62–69.

[32] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.. In *USENIX security symposium*. 569–584.