Knowl Inf Syst (2016) 49:143–169
DOI 10.1007/s10115-015-0903-y

CrossMark

REGULAR PAPER

A scalable approach for detecting plagiarized mobile applications

Ciprian Oprişa[1,2] · Dragoş Gavriluţ[1,3] · George Cabău[1,2]

Received: 16 April 2015 / Revised: 11 October 2015 / Accepted: 13 November 2015 /
Published online: 25 November 2015
© Springer-Verlag London 2015

Abstract Plagiarism cases are quite common in mobile applications ecosystems like the Android market. An application can be decompiled, modified and repackaged with a different author name. The modifications can affect the user's privacy or even contain malicious logic. If the original application is supported by advertisements, they are usually replaced so the ad revenue will go to the repackager. Such events can cause the legitimate author damage both in reputation and financially so they need to be detected. A plagiarism detection system is proposed that can detect plagiarized applications based on the features extracted from code. Two similarity functions are given along with techniques for finding similar applications in a large collection. The main issue with this search is that it cannot be performed sequentially, by comparing a given item with every other item in the collection. The built solution will improve the search time by comparing the searched item only with those with a high probability of being similar. The greatest advantage of our approach is scalability. The system's database can be built, updated and queried in reasonable time, even when large quantities of data are involved. Our experiments were conducted on a large collection of over one million samples and managed to identify a concerning number of plagiarism cases.

Keywords Plagiarism · Code similarity · Scalability · Clustering

✉ Ciprian Oprişa
coprisa@bitdefender.com

Dragoş Gavriluţ
dgavrilut@bitdefender.com

George Cabău
gcabau@bitdefender.com

1 Bitdefender, Cluj-Napoca, Romania

2 Technical University of Cluj-Napoca, Cluj-Napoca, Romania

3 Alexandru Ioan Cuza University of Iaşi, Iasi, Romania

1 Introduction

Smartphone sales experienced a spectacular growth, as they surpassed the sales of regular phones in 2013 [11]. The largest market share belongs to the Android devices, with more than 758 million units being sold last year. One of the causes for this success is the vast applications ecosystem, where users can chose from more than over one million apps [30].

These Android applications are usually written in Java [6] and run in custom environments [5] like Dalvik virtual machine or lately in Android Runtime (ART). The developers publish them on various markets, the most popular currently being the Play Store. For an entity to publish something in the Play Store it needs to perform a one-time payment of 25$. When an application is published, the publisher has the following revenue methods:

- in-app payment (usually used for games to buy premium items)
- the application itself can have a price
- the application can contain different Adware SDKs [2] (the most popular option)

One problem experienced by the developers is that their applications are easy to decompile and revert to the source code [19], just like any Java applications. There is currently no mechanism to prevent an attacker to plagiarize an existing application. The usual workflow for the attacker is depicted in Fig. 1:

(a) choose an application from the market. Usually a popular application is chosen (as this increases the probability for somebody to download the repackaged version). Android applications come as APK files, which are basically ZIP archives that contain the program's code, resources and manifest.

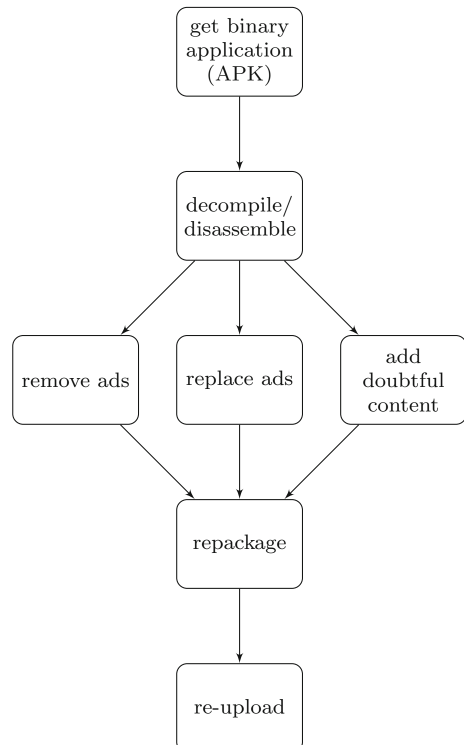

Fig. 1 App plagiarism workflow

(b) decompile or disassemble the application [1]
(c) modify some components
(d) recompile and sign the application with the new vendor key
(e) publish the application

This technique has several advantages for the attacker:

- He/she does not have to deal with the tedious job of writing an app/testing it/creating the graphics and so on, since all of these things are already done by somebody else.
- He does not need to promote the application—the vendors of the original app are already doing this.
- The most important aspect: it does not take too long to create a repackaged app. An experienced hacker could do it in less than an hour. This translates in small effort for a great applications and good revenue.

If the attacker's goal is to make profit from the plagiarized app, he will usually add new advertisement SDKs to it and remove the existing ones. Sometimes things can be as simple as replacing the author identification in the advertisement SDK in order to redirect the ad revenue to the attacker. The attacker may also add new content to the plagiarized application, but this content will not be for the user's benefit. They can add spyware code that affects the user's privacy or even malicious logic.

The difficulty of detecting such plagiarism cases consists in the fact that the new application code is similar but not identical with the original one. Besides the modification performed on the advertisement SDKs or code additions, the attacker may perform various obfuscation techniques at the repacking stage. Simple tools like ProGuard [16] can "remove unused code and rename classes, fields, and methods with semantically obscure names." More advanced obfuscation tools can perform code-level transformations, like adding garbage instructions or modifying blocks of code with semantically equivalent ones.

Since the names of the classes, fields and methods are easily renamed into something completely different, we will only consider the program's code when checking for similarity. We will use OpCodes n-grams here, as they proved robust enough to detect code similarities on other platforms [20,27]. In the context of this paper, an OpCode or *operation code* will be the operator part of an assembly instruction, without the operands. An OpCode n-gram is a sequence of n consecutive OpCodes extracted from a program's code. Since the application's code is separated into methods, we will also work with methods, by computing hash functions on each method's n-grams. We will use classic hashing for identifying identical methods and locality-sensitive hashing [14,24] for identifying similar methods (methods with a high percentage of common n-grams).

Unlike other platforms like Windows where most library code comes as external dynamical link libraries, an Android APK contains the library packages in the same file (`classes.dex`) as the application's code. Since the size of library code can exceed the size of application code, it is important to filter it out before computing the similarity score. Otherwise, a pair of different applications that use the same library may be flagged as similar.

The paper proposes two methods for computing the similarity between Android applications: shallow similarity and deep similarity. Shallow similarity will work at methods level, taking into account the common methods. Deep similarity will work at n-grams level and will also consider the near-identical methods in the applications.

For each approach, we will propose an algorithm that retrieves the similar items in a collection, without comparing the searched item with every other item. Although the two similarity functions are based on the same principle, this search will be different because the

number of n-grams has a greater order of magnitude than the number of methods so we will need to do an approximate search for the deep similarity.

The database containing the relevant information about applications, methods and n-grams will be constructed using a Map-Reduce approach [9]. This programming model takes as input a set of key-value pair and produces another set of key-value pairs. The programmer must supply a *map* function that requires an input pair and insures an intermediate key-value pair. The framework feeds all the intermediate pairs with the same key to a *reduce* function that produces the output. The biggest advantage of this model is the scalability, as it is able to process large amounts of data on thousands computing nodes.

The following section will present other approaches for finding plagiarism cases in mobile applications and is followed by a description of commonly used attack vectors for repackaging apps. The two code similarity function will be detailed next, followed by the architecture of a system that can store a continuously growing collection of applications and retrieve plagiarism cases. The next section will present the experimental results obtained by our system along with some of the practical issues that we have encountered. The paper ends with conclusions and future work directions.

2 Related work

Plagiarism detection has been approached in other papers as well. Any kind of data can be plagiarized, including documents, multimedia files or programs (both at source code level or at binary level). We are not interested in verbatim copies, as they are easy to spot by hashing an entire collection of documents and comparing the documents with the same hash. Near-identical copies, where the plagiarizer makes some modifications are harder to detect. Most approaches relied on the concept of n-grams (also called k-grams). An n-gram can be considered as a group of n consecutive items from a sequence.

A tool called *sif* was presented in [18] and was able to find similar files in a large file system. The authors considered two files to be similar if “they contain a significant number of common substrings that are not too small.” It was one of the first attempts to apply fingerprinting techniques in order to find matches in a large collection.

Heintze in [13] approached the scalability problem. The presented solution only takes into account a fixed number of fingerprints, in order to preserve storage space.

The ideas behind Moss, a publicly available solution is presented in [26]. The system is designed for plagiarism detection in programming assignments. The authors also host a web server where users can upload a collection of source codes and receive a similarity report. Moss also includes a visualization component that shows similar regions in the source code.

Mobile applications plagiarism detection is also an interesting research topic, as we identified several recent papers dealing with this subject.

In [22], the focus was on cases where the attacker added malicious code to existing applications. The authors showed that 29.4 % applications from a collection of 158,000 are more likely to be plagiarized because they already have the permissions that an attacker needs. For deciding whether two programs are similar or not, three schemes were proposed: *Symbol-Coverage*, *AST-Distance* and *AST-Coverage*. The first one relies on the symbol names extracted from the application and works only if no form of obfuscation has been involved. The other two schemes are based on *Abstract Syntax Trees*, a model that considers for each method only the number of arguments and the other invoked methods. Based on these trees, feature vectors can be built for the entire application or at method level. *AST-Distance* finds

plagiarism cases in a collection by comparing the feature vector extracted from a given application with the feature vectors of the other applications in the collection, while *AST-Coverage* matches the method-level feature vectors of two applications and searches for the maximum coverage. All these schemes work by comparing a given application with all the others in the collection. The tested collection was small, containing only 7600 samples.

The authors in [8] attempted to eliminate the pairwise similarity computations by clustering the applications in the first phase. The clustering is performed based on the application's attributes. Only the applications situated in the same cluster will then be pairwise verified for plagiarism by comparing the Program Dependence Graphs [10]. The graphs are compared in two phases. The first phase tries to filter out the pairs that are too different while the second one performs the more costly operation of finding subgraph isomorphisms. The paper also addresses the problem of eliminating the library code and the author identification so that two programs belonging to the same author won't be flagged as plagiarism. The framework was tested on a collection of 75,000 free Android applications and managed to check for plagiarism an average of 0.71 application pairs per minute.

Juxtapp is another system proposed by Hanna et al. [12] that detects vulnerable code reuse, malicious samples and piracy (plagiarism) cases. This system uses OpCode n-grams like ours but doesn't split the code into methods. Since the number of n-grams extracted from an application is quite large, a feature hash is produced, represented as a bit vector. Two such bit vectors can be compared using the Jaccard distance. Finding all cases of similar applications is still a quadratic problem, as each pair must be checked. The system was tested on a collection of 58,000 applications from the Android official market and from the Anzhi third party market.

The three systems above employ various methods for detecting whether two applications are similar but are working on small samples collections (less than 100,000). The focus of our system is scalability, as it manages to deal with a collection bigger than one million samples, while still correctly identifying plagiarism cases. Also, our paper presents some practical considerations when dealing with the applications ecosystem. One such consideration is that in many Android applications, the quantity of library code exceeds the quantity of application-specific code, making library code identification a top priority.

3 Attack vectors for plagiarizing mobile applications

This section will summarize the attack techniques used for plagiarizing Android applications. We will take into account the entire applications market environment, since a simple comparison between applications is not enough. For instance, we will assume that the goal of the attacker is to make profit. If the cost and effort for plagiarizing and publishing the cloned application is comparable to the cost and effort for developing a new legitimate one, the attack is not likely to happen. Even if this cost is a fraction (like 10 %) of the development cost for a new app, it is still more profitable to develop than to plagiarize, since cloned apps are usually short-lived (they are soon reported and taken down from the market). For instance, the case study in Sect. 6.2 shows that plagiarized applications managed to stay on the market up to 10–14 days. During this time, most of them got between 1000 and 5000 installs by users. However, the original application had between 1 and 5 million users (Google Play doesn't report exact numbers, only intervals), which is 1000 times better.

Another valid reason for reusing the code of an existing application is to spread malware. According to [31], 86 % of the malware samples are "repackaged versions of legitimate

applications with malicious payloads." This means that detecting repackaged applications will help discover new malware.

In 2007, Roy and Cordy wrote a comprehensive survey [25] on software cloning. Although their scope goes beyond Android and mobile applications, their taxonomy can successfully be applied in our case. The authors of the survey identified four types of cloning, the first three being based on textual similarity, while the last one considers functional similarity:

– Type I: "Identical code fragments except for variations in whitespace and comments." Since whitespace and comments are not considered when producing bytecode, no plagiarism detection system that relies on already compiled applications will be affected by this type.
– Type II: "Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments." Some plagiarism detectors take into account the name of methods, classes or packages (like the Symbol-Coverage technique from [22]). Official tools like ProGuard [16] allow developers to eliminate these types of literals from their applications in order to make them smaller and more resistant to reverse engineering. Our proposed system will not take into account the literal names, so it is unaffected by this type of cloning.
– Type III: "Copied fragments with further modifications." Besides cloning, code-level modifications are introduced manually or automatically. PANDORA [23] is a proof-of-concept tool designed to apply obfuscation techniques to the Android bytecode in order to make it different from a syntactic point of view. Depending on the obfuscation level, similarity functions based on n-grams or Abstract Syntax Tree are still able to detect common elements between the original and the plagiarized application. The difficulty mainly consists in balancing the true-positive and the false-positive rates. A small similarity threshold will detect more plagiarism cases but will also flag some legitimate apps as being plagiarized. A high similarity threshold will avoid most of the false positives but will also miss some of the clones.
– Type IV: "Two or more code fragments that perform the same computation but implemented through different syntactic variants." It is generally undecidable [29] if two programs produce the same output on any given input. Even human experts will not always agree if a pair of samples is a plagiarism case or just a reuse of the same idea.

Since the approach in this paper is based on the binary code found in the *classes.dex* file, it is not affected by Type I and Type II cloning. Sometimes the delimitation between Type IV and Type III clones might not be clear, as one would need to decide whether a piece of code is re-implemented or simply obfuscated. In what follows, we will consider Type IV changes only those changes performed by humans, while code-level changes performed by automated system will be classified as Type III, regardless of their sophistication. Type IV clones are difficult to detect using only statical analysis and difficult to prove. Although the general problem of identifying Type IV clones is equivalent to the halting problem [29], there are particular cases where automated analysis can still give valuable insights. If only some parts of the code are re-written, while the others are left intact or simply obfuscated, the remaining parts can still trigger a high similarity score. The quantity of manually re-written code is also correlated with the development cost for the attacker: Making few changes would be cheap, but the risk of being detected is high, while making too many manual changes would be impractical. Even if all the methods of an application are re-implemented, the clone can still be detected using structural features, if the classes structure and hierarchy are left unchanged.

In a 2012 paper [22], the authors also described two levels of obfuscation: level 1 only addresses changes to the symbol table (this corresponds to a type II cloning from Roy's taxon-

omy), while level 2 considers a number of added methods with no functionality (this partially covers type III). According to the authors, only level 1 obfuscation has been encountered in practice.

Detecting type III clones depends on the nature and the number of changes applied to the original code. The approach in this paper is based on OpCodes n-grams, a method robust to most of these changes. Bellow, we will enumerate the type III cloning approaches from [25] and see how the current approach handles them.

- *near-miss clones* non-identical code fragments that keep the syntactical structure. Since the OpCode includes only the operation to be performed, not the operands, a code fragment will be abstracted as a sequence of operations, that will remain identical, in this case.
- *gapped clones* and *non-contiguous clones* some code segments are inserted or deleted. The success of the n-grams approach is highly dependent on the amount of changes. If enough segments of at least n consecutive instructions are preserved, the clone will still be detected. If a significant amount of code is altered this way, the clone will evade detection, but the effort for the attacker will also be higher. A clone with a large number of changed code fragments may also be categorized as type IV clone, which is out of our scope.
- *structural clones* and *function clones* the syntactic structure of the program is modified. In the Android case, some classes can be moved from one package to another, or some methods can be moved from one class to another. Also, package and classes can be split or joined. Since our abstraction considers an application to be a set of methods, such structural changes do not modify the abstract view. The only change that can affect our approach is to move blocks of code between methods.
- *reordered clones* the order of some code segments is changed, such that the code semantics is preserved. Two methods will be compared as unordered sets of n-grams. The only n-grams that will differ will be the ones that cross the border between two segments. For instance, if the sequence of operations $ABCDEFGH$ is changed into $EFGHABCD$ (the two halves are swapped), and we extract 2-grams ($n = 2$), the only different 2-g will be DE that will be replaced by HA, while the remaining 2-g (AB, BC, CD, EF, FG and GH) will remain the same.

The applications similarity functions in the next section will output a similarity score, based on the amount of common code, between two applications. Since type I and II clones produce identical features, the similarity score will be 100 %, so there will be no issue in detecting them. For type III clones, however, the similarity threshold (the lower bound, above which a pair of applications will be suspected as plagiarism) must be chosen such to maximize detection while avoiding false positives. The problem is that many Android applications contain a large amount of library code. It is not uncommon for an Android application to be comprised of more than 90 % (sometimes even more than 99 %) library code. Two different applications may look very similar because they are using the same library, so even if we set the threshold at 90 %, we can't avoid the false positives.

Our approach will try to identify library code and disregard it while computing the similarity score. Some library code was labeled manually, while most of it was identified because it appeared in multiple applications from different developers. A possible attack against this method is to publish multiple plagiarized versions of the same application under different developer IDs. If enough different versions will be found in the market, an automated system will automatically label the entire application code as library and will not identify further plagiarism cases. This attack is viable, since creating a new developer account is relatively inexpensive (25$). A mitigation to this attack is to assign a reputation score to each developer, based on the number of application he published, the user ratings for those applications and

the amount of time he has been in the market. A code fragment will only be flagged as library if the sum of the reputation values for the developers that use it exceeds a threshold. This will not make the attack impossible, but will increase the cost for the attacker enough for the plagiarism not to be profitable.

4 Applications similarity functions

This section deals with the similarity issue and proposes two methods for computing the similarity between applications: shallow similarity and deep similarity. Both methods are based on the applications code and use the concept of n-grams discussed in the introductory section.

A similarity function is a function that takes as input two applications and outputs a real number between 0 and 1, as in Eq. 1. If $sim(A_1, A_2) = 1$ it means that the two applications have the same non-library code, while a similarity of 0 means that they are completely different (except for library code).

$$sim : \mathcal{A} \times \mathcal{A} \rightarrow [0, 1] \tag{1}$$

Definition 1 We will say that two applications $A_1, A_2 \in \mathcal{A}$ represent a **plagiarism case** iff $sim(A_1, A_2) \geq \theta_p$. $\theta_p \in [0, 1]$ is a constant called *plagiarism threshold*.

The set $\mathcal{A}$ is the set of all applications in a collection. An application is represented as a set of methods: $A = \{M_1, M_2, \ldots, M_k\}$ $M_i \in \mathcal{M}, \forall i \in \overline{1, k}$. The set $\mathcal{M}$ is the set of all the unique methods found in the collection's applications. We will also represent each method as a set of n-grams. An n-gram, as discussed in the first section, is a sequence of n consecutive OpCodes from a method. The set of all n-grams from the collection's methods will be denoted by $\mathcal{G}$.

4.1 Publishers identification and library code

In order to publish an application in the Android market developers need to create an account [3]. Currently, there is a one-time fee of 25$ for creating a new account, "to encourage higher quality products on Google Play (i.e., less products with SPAM)." A developer may publish as many applications as he wants from the same account. An application is uniquely identified by a package name and is self-signed with a "certificate whose private key is held by the developer" [4]. In order to update an existing application, the new version must have the same package name and be signed with the same certificate.

Since the publisher name can be easily changed in the developer's account, we will rely on the following two facts when identifying the publishers:

- If two applications have the same package name, they belong to the same publisher.
- If two applications are signed with the same certificate, it means that the signers had the same private key so they must be the same entity.

The above assumptions don't hold if the private key of a publisher is leaked or if the certificate's algorithm is insecure. On this matter, we have identified several applications that are still signed using the MD4 hash algorithm that was already proved to be insecure [17]. However, most applications use MD5 or SHA for hash in their certificate.

In what follows, we will consider $pub : \mathcal{A} \rightarrow \mathbb{N}$, a function that assigns a publisher id to each application. If $pub(A_1) = pub(A_2)$, it means that the applications A_1 and A_2 have the same publisher.

We will now give the definition for *library code*. Informally, we will consider library code a piece of code that is widely used, usually by many publishers. To improve the system's accuracy, some n-grams and methods may be manually marked as library code. The following definitions will formally illustrate the concepts of library n-grams and library methods.

Definition 2 The set of library n-grams is a set $\mathcal{G}_L \subset \mathcal{G}$ that contains all the n-grams found in at least $\theta_{\mathcal{G}L}$ different methods and the n-grams manually marked as library code, as in Eq. 2. The set of non-library n-grams will be denoted by $\mathcal{G}^\star = \mathcal{G} \backslash \mathcal{G}_L$.

$$\mathcal{G}_L = \{g \in \mathcal{G} \mid |\{M \in \mathcal{M} \mid g \in M\}| \geq \theta_{\mathcal{G}L}\} \cup L_{\mathcal{G}} \tag{2}$$

$\theta_{\mathcal{G}L}$ is a chosen threshold for library n-grams and $L_{\mathcal{G}}$ is the set of manually chosen library n-grams. For performance reasons, we won't associate the n-grams with publishers and will relay only on the number of methods they appear into.

Definition 3 The set of library methods is a set $\mathcal{M}_L \subset \mathcal{M}$ that contains all the methods found in at least $\theta_{\mathcal{M}L1}$ different applications or used by at least $\theta_{\mathcal{M}L2}$ publishers and the methods manually marked as library code, as in Eq. 3. The set of non-library methods will be denoted by $\mathcal{M}^\star = \mathcal{M} \backslash \mathcal{M}_L$.

$$\begin{aligned}\mathcal{M}_L =&\{M \in \mathcal{M} \mid |\{A \in \mathcal{A} \mid M \in A\}| \geq \theta_{\mathcal{M}L1}\} \\ &\cup \{M \in \mathcal{M} \mid |\{pub(A) \mid A \in \mathcal{A} \wedge M \in A\}| \geq \theta_{\mathcal{M}L2}\} \\ &\cup L_{\mathcal{M}}\end{aligned} \tag{3}$$

$\theta_{\mathcal{M}L1}$ and $\theta_{\mathcal{M}L2}$ are a chosen thresholds for library methods and $L_{\mathcal{M}}$ is the set of manually chosen library methods.

In order to mitigate the attack described in the previous section, where a plagiarized application can be re-published using multiple accounts so that the entire code will be flagged as library code, we can alter Eq. 3 in the following way:

$$\begin{aligned}\mathcal{M}_L =& \{M \in \mathcal{M} \mid |\{A \in \mathcal{A} \mid M \in A\}| \geq \theta_{\mathcal{M}L1}\} \\ &\cup \left\{ M \in \mathcal{M} \;\middle|\; \sum_{P \in \{pub(A) \mid A \in \mathcal{A} \wedge M \in A\}} \text{reputation}(P) \geq \theta_{\mathcal{M}L2} \right\} \\ &\cup L_{\mathcal{M}}\end{aligned} \tag{4}$$

Basically, in Eq. 4, instead of counting the number of distinct publishers, we add their reputation values. The reputation function is computed for a given publisher based on the user ratings and the number of downloads for their published apps. The cost for creating a large number of high-rated user accounts is high enough to discourage this attack. The threshold $\theta_{\mathcal{M}L1}$ is larger than the highest number of applications published by a single developer. This way, an attacker cannot force our system to flag all the application's methods as library code by publishing $\theta_{\mathcal{M}L1}$ versions of it using a single account. In theory, it is possible to get beyond this threshold using several accounts and publish a large number of clones with all of them, but this kind of behavior may raise other suspicions.

In the rest of this paper we will consider that each application in our collection has a reasonable number of non-library methods. An application with too few non-library methods (less than a given threshold) will be excluded from the collection $\mathcal{A}$ as we cannot find similar applications based on its extracted code.

4.2 Shallow similarity

After formally defining the concept of library code in the previous subsection, we can now define the shallow similarity between two applications in Eq. 5.

$$ssim : \mathcal{A} \times \mathcal{A} \rightarrow [0, 1]$$
$$ssim(A_1, A_2) = \frac{|\{M \in \mathcal{M}^\star \mid M \in A_1 \wedge M \in A_2\}|}{|\{M \in \mathcal{M}^\star \mid M \in A_1 \vee M \in A_2\}|} \quad (5)$$

In other words, the shallow similarity is the Jaccard similarity between the sets where the library methods have been filtered out. It computes the ratio between common non-library methods and total non-library methods.

Let $A \in \mathcal{A}$ be an application. If we denote by $A^\star \subset A$ the subset of non-library methods from A, $A^\star = \{M \in A \mid M \in \mathcal{M}^\star\}$, Eq. 5 can be rewritten in a simpler way, in Eq. 6.

$$ssim(A_1, A_2) = \frac{|A_1^\star \cap A_2^\star|}{|A_1^\star \cup A_2^\star|} \quad (6)$$

One disadvantage of this similarity function is that it assumes the attacker will leave most of the application's code unaltered. If the attacker uses an obfuscation tool that slightly modifies the code of each method so at least an n-gram will be altered, the shallow similarity will fail to recognize the plagiarism. Internally, a method is stored as a hash on the sorted list of n-grams, so the method hashes won't be the same.

In [22] it is argued that although advanced obfuscation methods exist, "their applicability to mobile applications remains unknown due to the specific byte-code format and the tight resource and energy constraints." Their model allows the attacker to add or remove some methods but not to perform modifications at the method level.

For the cases where method-level modifications are performed, our system is still able to detect the plagiarism cases, by using the deep similarity presented in the next subsection.

If the average number of non-library methods in an application is m (we have already denoted with n the number of consecutive OpCodes in the n-grams) and the applications are represented as sorted lists, the computation complexity is $O(m)$.

4.3 Deep similarity

Deep similarity goes beyond method level and tries to match the non-identical methods between the applications based on their n-grams.

First of all, we will define the similarity between two methods in the same way the shallow similarity between two applications was defined.

$$msim : \mathcal{M}^\star \times \mathcal{M}^\star \rightarrow [0, 1]$$
$$msim(M_1, M_2) = \begin{cases} 0, & \text{if } |\{g \in \mathcal{G}^\star \mid g \in M_1 \cap M_2\}| = 0 \\ \dfrac{|\{g \in \mathcal{G}^\star \mid g \in M_1 \cap M_2\}|}{|\{g \in \mathcal{G}^\star \mid g \in M_1 \cup M_2\}|}, & \text{otherwise} \end{cases} \quad (7)$$

The $msim$ function in Eq. 7 computes the similarity between two methods using the Jaccard similarity function on the non-library n-grams. Notice that a method may have all the n-grams in $\mathcal{G}_L$ (library n-grams) yet not be a library method. If we need to compute the similarity between two such methods, the denominator of the fraction would be 0. For this reason, the first branch of the equation states that if the two methods have no common n-grams, their similarity is be 0.

For practical purposes, we may consider the similarity between two methods only if it is above a certain threshold (θ_m). In this case, we will use $msim'$ instead of $msim$:

$$msim'(M_1, M_2) = \begin{cases} msim(M_1, M_2), & \text{if } msim(M_1, M_2) \geq \theta_m \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

If we take a closer look at Eqs. 5 and 6, we observe that the difference between the numerator and the denominator of the fractions consists in the non-common methods from both A_1 and A_2. Equation 6 can be further rewritten as:

$$ssim(A_1, A_2) = \frac{|A_1^\star \cap A_2^\star|}{|A_1^\star \cap A_2^\star| + |A_1^\star \backslash A_2^\star| + |A_2^\star \backslash A_1^\star|} \tag{9}$$

Because shallow similarity works at method level, it won't take into account matches between methods from $A_1^\star \backslash A_2^\star$ and methods from $A_2^\star \setminus A_1^\star$ that have common n-grams.

Given two sets of methods $X, Y \in \mathcal{M}^\star$, with $X \cap Y = \emptyset$, we can associate each pair (x, y) with $x \in X$, $y \in Y$ with a weight, $w(x, y) = msim(x, y)$. At this point, we can find a matching between the two sets of methods by solving the **maximum weighted bipartite matching** problem (also called the assignment problem). Several polynomial algorithms exist for this task, the most notable one being the Hungarian algorithm [15].

The best match value between the sets X and Y will be denoted by $bm(X, Y)$ and is expressed in Eq. 10.

$$\begin{gathered} bm : \mathcal{M}^\star \times \mathcal{M}^\star \rightarrow \mathbb{R}_+ \\ bm(X, Y) = \max \sum_{i=1}^{\min(|X|,|Y|)} msim'(x_i, y_i), \\ x_i \in X, y_i \in Y, \\ x_i \neq x_j, y_i \neq y_j, \forall i \neq j \end{gathered} \tag{10}$$

The best match value between the non-common methods of the two applications A_1 and A_2 can be added to the fraction's numerator in Eq. 6 to compensate for the non-identical function matches. This new similarity function will be called deep similarity and will be expressed in Eq. 11.

$$\begin{gathered} dsim : \mathcal{A} \times \mathcal{A} \rightarrow [0, 1] \\ dsim(A_1, A_2) = \frac{|A_1^\star \cap A_2^\star| + 2 \cdot bm(A_1^\star \backslash A_2^\star, A_2^\star \backslash A_1^\star)}{|A_1^\star \cup A_2^\star|} \end{gathered} \tag{11}$$

The best match value at the numerator is scaled by 2, because each pair of non-common methods $M_1 \in A_1$, $M_2 \in A_2$ contributes with 2 to the denominator while their match can contribute to the numerator with less than 1.

Unlike the shallow similarity, deep similarity is able to identify plagiarism cases even if the attacker performs changes in the code of each method. The drawback is that the deep similarity function is harder to compute. If the average number of non-library methods in an application is m, the complexity of the Hungarian algorithm that finds the best match is $O(m^3)$, which is considerably higher than $O(m)$, the cost for computing the shallow similarity.

5 System's architecture and algorithms

The previous section presented the two similarity functions for Android applications. Shallow similarity is faster to compute while deep similarity is more robust to obfuscations and both of

them work well when they need to check a pair of applications for plagiarism. Unfortunately, the real-world problems are more difficult than this. One common use-case is to add a new application to the system and search for all similar applications belonging to different publishers. Another one would be to output all the plagiarism cases found in the collection. The system also need to scale well and deal with a collection of millions of applications.

5.1 Data model

In order to satisfy the scalability requirement we have used a NoSQL database. Our choice was MongoDB [7]. The data are stored into four main collections:

- *ApkToMet* maps the relation between applications and methods. Each item is a document that stores the application id and the sorted list of method hashes in the application. The application's publisher id is also stored here.
- *MetToNgr* maps the relation between methods and n-grams. For each method, we will store a sorted list with all the method's n-grams.
- *MetToApk* is the reversed index for the *ApkToMet* collection. If a method is non-library, the collection will hold all the applications that contain it. Otherwise, a boolean field of the document will store the fact that this is a library method.
- *NgrCount* is a simple collection that counts how many times each n-gram appears in methods.

The reversed index (*MetToApk* collection) is used for finding similar applications without checking every item in the *ApkToMet* collection.

Instead of storing a mapping form n-grams to methods, the last collection will store a simple count for each n-gram. Storing an entire list of methods for each n-gram would take up too much space. We will show that we can still find methods similar to a given one without using a reversed index, by employing the locality-sensitive hashing technique.

The ER (Entity-Relationship) diagram is presented in Fig. 2.

5.2 Database construction

Building the database presented in the previous subsection from a raw collection of applications is no easy task, due to the size of the data. We will use the Map-Reduce model [9] for dividing this task into smaller tasks.

This model involves two functions: MAP and REDUCE. MAP takes as input a raw data item and produces key-value pairs, through the function EMIT. The Map-Reduce framework groups all these pairs by the key and then passes to each reducer a key and the list of values that were emitted for that key. The REDUCE function should implement the processing of these value lists for each key. After the reduce phase, the algorithm may finish or use the results for another map-reduce stage.

Our implementation uses two map-reduce stages for building the entire database, as in Fig. 3.

In the first stage, each application is processed by the function MAP-1 described in Algorithm 1.

MAP-1 processes an application and extracts the methods. For each method, a hash on the sorted list of n-grams is computed by the HASH function and added to the $methodHashes$ list (line 3). Each method is also emitted along with the id of the current application (line 4). After the list of method hashes has been constructed, it is inserted into the *ApkToMet* collection with the application's id as primary key.

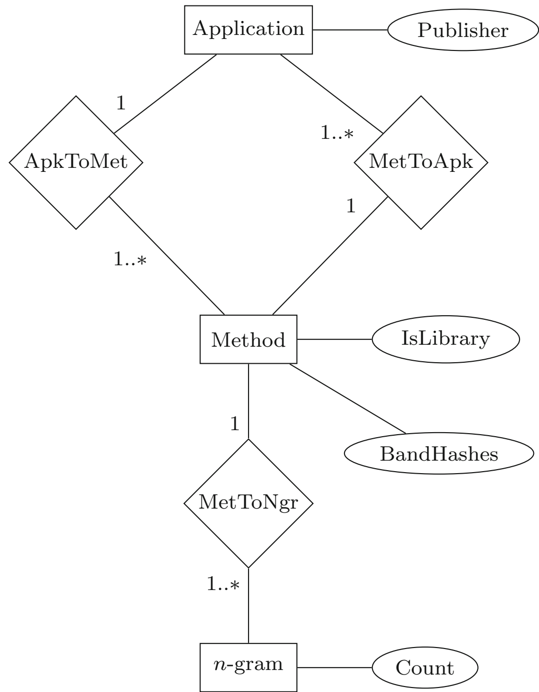

Fig. 2 Data model ER diagram

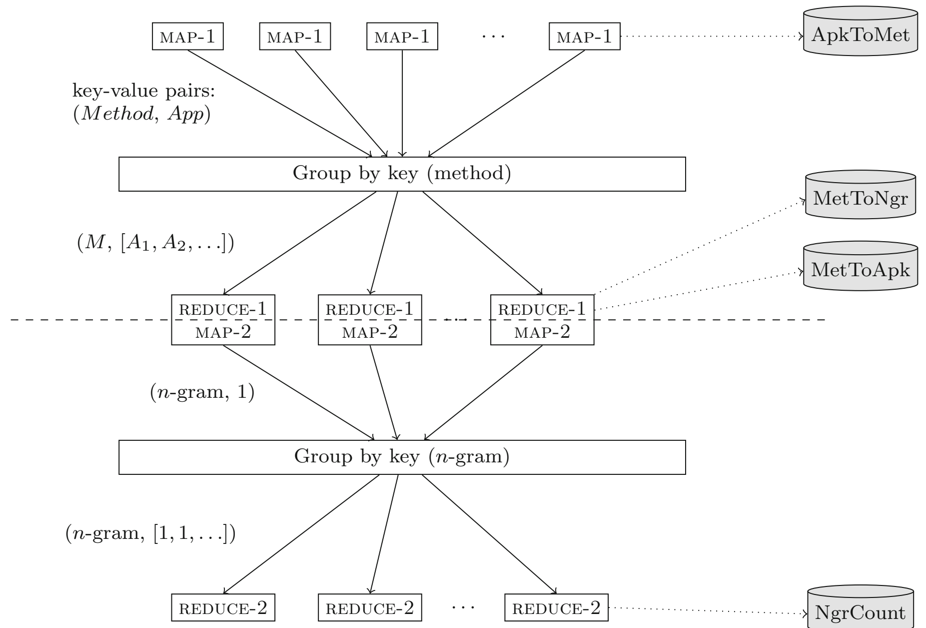

Fig. 3 Database building

Algorithm 1 MAP-1(A)

Require: An application $A \in \mathcal{A}$
Ensure: A will be stored in the *ApkToMet* collection
Ensure: All methods will be emitted.

1: $methodHashes \leftarrow \{\}$
2: **for all** $M \in A$
3: $\quad methodHashes \leftarrow methodHashes \cup \{\text{HASH}(M)\}$
4: $\quad \text{EMIT}(M, A.id)$
5: **end for**
6: $ApkToMet.\text{INSERT}(A.id, methodHashes)$

The Map-Reduce framework groups together all the application ids emitted for the same method. The reducers from the first stage, implemented by the function REDUCE-1 in Algorithm 2, will receive a method with all the emitted application ids for it.

Algorithm 2 REDUCE-1$(M, Apps)$

Require: A method $M \in \mathcal{M}$
Require: A list of application ids $Apps$
Ensure: The method's n-grams will be stored in the *MetToNgr* collection.
Ensure: The method M will be stored in *MetToApk* collection.

1: $MetToNgr.\text{INSERT}(\text{HASH}(M), M, \text{BAND- HASHES}(M))$
2: **if** $|Apps| \geq \theta_{\mathcal{M}L1}$ **or** $|\text{UNIQUE- PUBS}(Apps)| \geq \theta_{\mathcal{M}L2}$ **then**
3: $\quad MetToApk.\text{INSERT}(\text{HASH}(M), IsLibrary = 1)$
4: **else**
5: $\quad MetToApk.\text{INSERT}(\text{HASH}(M), Apps = Apps)$
6: **end if**
7: **for all** $g \in M$
8: $\quad \text{EMIT}(g, 1)$
9: **end for**

Algorithm 2 starts by inserting the method's n-grams in the *MetToNgr* collection, with the method hash as primary key. The reason we perform the insertion at this point and not in the MAP-1 function is that the same method may belong to several applications. A unique index would ensure it won't be inserted more than once, but the search would still be performed every time. The band hashes computed on the method's n-grams are also inserted in the database at this point, in order to ensure that similar methods can be retrieved given a set of n-grams. The locality-sensitive hashing approach will be detailed in Sect. 5.3.3.

Next, the method is inserted in the *MetToApk* collection. If the method is a library method ($M \in \mathcal{M}_L$), according to Definition 3 and Eq. 3, the list of application ids won't be stored into the collection, just the information that it's a library method (line 3).

According to Fig. 3, the reduce phase in the first stage (REDUCE-1) is immediately followed by the map phase from the second map-reduce stage (MAP-2). Since both operations work at the method level, they have been joined in Algorithm 2. The MAP-2 operation is performed in the lines 7–9, and for each n-gram $g \in M$, the value 1 is emitted for the key g. This ensures that the reducers from the second stage will count the number of containing methods for each n-gram.

The second-stage reducers are described by the function REDUCE-2 from Algorithm 3 and perform a single operation: count how many times an n-gram was emitted and insert this information into the *NgrCount* collection. Since each time an n-gram is emitted the value 1

Algorithm 3 REDUCE-2(g, Lst)

Require: An n-gram $g \in \mathcal{G}$
Require: A list of 1s Lst
Ensure: g will be stored in *NgrCount* collection.

1: $NgrCount$.INSERT(g, $|Lst|$)

is used, the parameter Lst of REDUCE-2 will be a list containing as many values of 1 as the number of methods the n-gram g was found in.

5.3 Searching for similar items

We have established in the previous section that both shallow and deep similarity are easy to compute when a pair of samples needs to be checked for plagiarism. The problem gets more difficult when we are searching for all the applications in the collection that are similar to a given one or when we are searching for all the pairs of similar applications. We would like to avoid the naive algorithms where we have to compare the searched application with every other application in the first case or check every pair of applications in the second case. This subsection will propose three new algorithms for:

- finding similar items based on shallow similarity
- finding all pairs of similar items based on shallow similarity
- finding similar items based on deep similarity

Each of the following algorithms is based on the Map-Reduce model. Local MAP and REDUCE procedures will be implemented in each one of them. As discussed above, MAP will receive a raw item and emit key-value pairs, while REDUCE will be called for each key and the associate list of values and optionally return an output. The PERFORM- MAP- REDUCE function puts it all together by calling MAP on each item from the input list and return a list with all the outputs returned by the reducers. Unlike classical Map-Reduce frameworks, we have designed our system to perform fast similarity queries so fault tolerance was not an issue. This gives a better performance, since there is no need to store intermediary results on permanent storage and more flexibility, since the MAP and REDUCE functions need not to be stateless.

5.3.1 Finding similar apps based on shallow similarity

Algorithm 4 receives an application $A \in \mathcal{A}$ and produces a list with all the similar applications that have different publishers than A, based on shallow similarity.

The MAP function is called for each method $M \in A$. The method is searched in the *MetToApk* collection (line 2). If M is a non-library method, then each application B in its list of apps is a candidate for similarity so it will be emitted (line 5).

A candidate B will be emitted once for each non-library method that is common with A, so the number of elements in the list Lst, the second parameter of REDUCE will be equal to this number: $|Lst| = |A^\star \cap B^\star|$. To compute the shallow similarity as in Eq. 6, we will also need the size of the union, for the fraction's denominator. By applying the inclusion–exclusion principle, we have:

$$\begin{aligned} |A^\star \cup B^\star| &= |A^\star| + |B^\star| - |A^\star \cap B^\star| \\ &= |A^\star| + |B^\star| - |Lst| \end{aligned}$$

Algorithm 4 FIND- SIMILARS- SHALLOW(A)

Require: An application $A \in \mathcal{A}$
Ensure: A list of applications similar with A

```
 1: procedure MAP(M)
 2:    Rec ← MetToApk.QUERY(M)
 3:    if not Rec.IsLibrary then
 4:       for all B ∈ Rec.Apps
 5:          EMIT(B, 1)
 6:       end for
 7:    end if
 8: end procedure
 9: procedure REDUCE(B, Lst)
10:    s ← |Lst| / (|A*| + |B*| − |Lst|)
11:    if s ≥ θ_p and pub(A) ≠ pub(B) then
12:       return B
13:    end if
14: end procedure
15: return PERFORM- MAP- REDUCE(A)
```

This means that the equation in line 10 correctly computes the shallow similarity between A and B ($s = ssim(A, B)$). All we have to do now is to check whether this similarity is above the plagiarism threshold and that A and B have different publishers. If both conditions hold, B will be added to the list of applications similar with A.

To compute the complexity of Algorithm 4, we will assume that a database query takes constant time (if it is indexed properly) and that an average application has m methods. If $M \in \mathcal{M}^\star$, the maximum number of applications in a *MetToApk* record is $\theta_{\mathcal{M}L1}$, so the MAP function will emit at most $\theta_{\mathcal{M}L1}$ values. Since MAP is called for each method, the number of operations performed by the first phase of Map-Reduce is $O(m \times \theta_{\mathcal{M}L1})$. In the reduce phase, we need to compute $|B^\star|$, or how many non-library methods have the set B ($|A^\star|$ can be computed once then cached), so m database queries will be performed. In the worst-case scenario, the REDUCE function is called for each emitted pair, so it will be called $O(m \times \theta_{\mathcal{M}L1})$ times. Assuming that $\theta_{\mathcal{M}L1}$ is constant, the final complexity will be $O(m^2)$.

5.3.2 Finding all pairs of similar apps based on shallow similarity

Algorithm 5 builds a list with all the plagiarism cases found in the database and is very similar to Algorithm 4. It also has a map phase that processes methods, but the inputs won't belong to a single application but will be the entire *MetToApk* collection (or the set $\mathcal{M}$). For a non-library method $M \in \mathcal{M}^\star$, each pair of applications that contain it will be emitted with the value 1, for counting (line 5).

The REDUCE function is identical with the one from Algorithm 4, with the exception that the key input is a pair, not a single application and the output list will also contain pairs.

In order to speed-up, the computations, $|A^\star|$ and $|B^\star|$, won't be computed for every reducer. Instead, the number of non-library methods for each application can be precomputed (also with a Map-Reduce algorithm) and accessed in $O(1)$. The MAP function now considers all the pairs of apps that contain the non-library method $M \in \mathcal{M}^\star$, which number is at most $\frac{\theta_{\mathcal{M}L1}(\theta_{\mathcal{M}L1} - 1)}{2}$ or $O(\theta^2_{\mathcal{M}L1})$. The map phase that needs to call MAP $|\mathcal{M}|$ times is the

Algorithm 5 FIND- ALL- SIMILARS- SHALLOW()

Require: The database described in the previous subsection
Ensure: $AllSims$, a list of plagiarism cases

```
1: procedure MAP(M)
2:    Rec ← MetToApk.QUERY(M)
3:    if not Rec.IsLibrary then
4:       for all A, B ∈ Rec.Apps, A.id < B.id
5:          EMIT((A, B), 1)
6:       end for
7:    end if
8: end procedure

9: procedure REDUCE((A, B), Lst)
10:    s ← |Lst| / (|A*| + |B*| − |Lst|)
11:    if s ≥ θ_p and pub(A) ≠ pub(B) then
12:       return (A, B)
13:    end if
14: end procedure

15: return PERFORM- MAP- REDUCE(𝓜)
```

most costly part of the algorithm, since both the reduce phase and the counting of non-library methods for all applications take less operations. The complexity for Algorithm 5 is then $O(|\mathcal{M}| \times \theta^2_{\mathcal{M}L1})$ or $O(|\mathcal{M}|)$ if we consider $\theta_{\mathcal{M}L1}$ to be a constant.

5.3.3 Finding similar apps based on deep similarity

This algorithm is split into three parts. Algorithm 6 finds similar methods with a given one, while Algorithm 7 and Algorithm 8 implement the map and reduce phases for finding similar applications based on deep similarity.

The function FIND- SIMILARS- METHODS from Algorithm 6 receives a method M and outputs a list of methods whose similarity with M is above the threshold θ_m (from Eq. 8). Each item in the list is a pair that contains the similar method and the actual similarity score. The list is sorted by this similarity in descending order (the method with the highest similarity being the first).

The idea for finding similar methods is different from finding similar applications, because we don't have a reverse index for the *MetToNgr* collection. We will use locality-sensitive hashing [14] instead. Informally speaking, a locality-sensitive hash is a hash function where the collision probability for similar items is higher than the collision probability for dissimilar ones.

In [24] it is proved that the collision probability for a *minhash* is equal to the Jaccard similarity. A *minhash* is a function that retrieves the minimum element from a set, according to a permutation σ: $h(X) = \min_{x \in X} \sigma(x)$. A permutation σ on the set $\{0, 1, 2, \ldots p-1\}$ can be approximated by $\sigma(x) = (a \cdot x + b) \mod p$. The collision probability can be augmented with the banding technique: We will use $b \times r$ different minhash functions, grouped on b bands, each containing r rows. On each band, a band hash is computed on the results of the r minhash functions. If the Jaccard similarity between two applications is s, the probability that at least one band hash has the same value is $1 - (1 - s^r)^b$. Guidelines for choosing the parameters b and r were given in [21].

Let $\theta_m = 0.8$. If we pick $r = 4$ and $b = 10$, the probability for two methods with similarity θ_m to have at least one common band hash is 99.48 %. If the similarity between the two methods is 0.3, the probability drops to 7.81 %.

In order to be able to find similar methods with a given one, we will augment the collection *MetToNgr* with the b band hashes computed on the non-library n-grams. These band hashes need to be updated periodically, because an n-gram can move from $\mathcal{G}^\star$ to $\mathcal{G}_L$ after adding new applications to the collection. If we keep indexes on all these b band hashes, we can find similar candidates by querying the collection for items with the same band hash on each band.

Algorithm 6 FIND- SIMILARS- METHODS(M, A)

Require: A method $M \in A$
Ensure: A list of methods similar with M that don't belong to A and the associated similarities

1: $candidates \leftarrow \emptyset$
2: $MRec \leftarrow MetToNgr.$QUERY$(M)$
3: **for** $i = 1 \rightarrow b$
4: $\quad rs \leftarrow MetToNgr.$QUERY$(band_i = MRec.band_i)$
5: $\quad candidates \leftarrow candidates \cup rs$
6: **end for**
7: $results \leftarrow \emptyset$
8: **for all** $C \in candidates \setminus A$
9: $\quad s \leftarrow msim(M, C)$
10: $\quad$ **if** $s \geq \theta_m$ **then**
11: $\quad\quad results \leftarrow results \cup \{(C, s)\}$
12: $\quad$ **end if**
13: **end for**
14: SORT$(results)$
15: **return** $results$

Algorithm 6 begins by querying the database for the current method record. A list of candidate similar methods is produced, by querying the collection *MetToNgr* for records with the same band hashes (lines 2–6). Each candidate except for the actual queried method is checked for similarity with M and if the similarity is at least θ_m, the candidate and the similarity scores are added to the results list. Finally, the result list is sorted in descending order on the similarity field (line 14) and returned (line 15).

The function FIND- SIMILARS- METHODS that finds methods similar to the given parameter M is a useful tool for finding similar applications. The function that performs this task FIND- SIMILARS- DEEP is split between Algorithm 7 where we can find the map phase and Algorithm 8 that contains the reduce code.

The MAP function from Algorithm 7 receives one of the application's methods ($M \in A$) as a parameter and emits applications that are likely to be similar to A, along with a similarity weight and the information that the method is common with A or is just similar to one of A's methods.

If the method M is not a library method, each application B that contains it is emitted with the weight $w = 1$ and the information that the method is common to A and B, $common =$ **true** (line 5). For each method similar to M that doesn't belong to A and is not a library method, all applications that haven't been used so far for this method are emitted with the weight $w = 2 \cdot s$ and the information that the method is not common to A, $common =$ **false**.

The reason for choosing the weight to be twice the similarity is because the best match value is also multiplied by two in Eq. 11. The *common* field will be used by the REDUCE

function to treat differently the fact that an emit operation was performed from a common method or a similar method.

Algorithm 7 FIND- SIMILARS- DEEP(A)—part 1

Require: An application $A \in \mathcal{A}$
Ensure: A list of applications similar with A

1: **procedure** MAP(M)
2: $\quad Rec \leftarrow MetToApk$.QUERY($M$)
3: $\quad$ **if not** $Rec.IsLibrary$ **then**
4: $\quad\quad$ **for all** $B \in Rec.Apps \setminus \{A\}$
5: $\quad\quad\quad$ EMIT(B, ($w = 1$, $common =$ **true**))
6: $\quad\quad$ **end for**
7: $\quad\quad usedApps \leftarrow Rec.Apps$
8: $\quad\quad sims \leftarrow$ FIND- SIMILARS- METHODS(M, A)
9: $\quad\quad$ **for all** $(M', s) \in sims$
10: $\quad\quad\quad Rec \leftarrow MetToApk$.QUERY($M'$)
11: $\quad\quad\quad$ **if not** $Rec.IsLibrary$ **then**
12: $\quad\quad\quad\quad$ **for all** $B \in Rec.Apps$
13: $\quad\quad\quad\quad\quad$ **if** $B \notin usedApps$ **then**
14: $\quad\quad\quad\quad\quad\quad$ EMIT(B, ($w = 2 \cdot s$, $common =$ **false**))
15: $\quad\quad\quad\quad\quad\quad usedApps \leftarrow usedApps \cup \{B\}$
16: $\quad\quad\quad\quad\quad$ **end if**
17: $\quad\quad\quad\quad$ **end for**
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end for**
20: $\quad$ **end if**
21: **end procedure**

Algorithm 8 FIND- SIMILARS- DEEP(A)—part 2

1: **procedure** REDUCE(B, Lst)
2: $\quad num \leftarrow 0, common \leftarrow 0$
3: $\quad$ **for all** $val \in Lst$
4: $\quad\quad num \leftarrow num + val.w$
5: $\quad\quad$ **if** $val.common$ **then**
6: $\quad\quad\quad common \leftarrow common + 1$
7: $\quad\quad$ **end if**
8: $\quad$ **end for**
9: $\quad s \leftarrow \frac{num}{|A^\star| + |B^\star| - common}$
10: $\quad$ **if** $s \geq \theta_p$ **and** $pub(A) \neq pub(B)$ **then**
11: $\quad\quad s' \leftarrow dsim(A, B)$
12: $\quad\quad$ **if** $s' \geq \theta_p$ **then**
13: $\quad\quad\quad$ **return** B
14: $\quad\quad$ **end if**
15: $\quad$ **end if**
16: **end procedure**

17: **return** PERFORM- MAP- REDUCE(A)

Algorithm 8 describes the second phase of the FIND- SIMILARS- DEEP function, the reducers code. The inner REDUCE function is called for each application B that has a chance of being similar to A and also receives all the emitted values for it. At this point we could compute the value $dsim(A, B)$ and decide whether the applications are indeed similar or not. The

problem is that this computation is costly, so we want to perform it only for applications that are very likely to be similar to A.

The denominator of the fraction in Eq. 11 is easy to compute using the inclusion–exclusion principle like we did in Algorithm 4 because we can count exactly how many common methods were emitted from the *common* filed of each emitted value (lines 5–7). Next, we will prove the following lemma:

Lemma 1 *The sum of all the emitted weights for a key B in Algorithm 7 is greater or equal than the fraction's numerator in Eq. 11.*

Proof Each time the application B is emitted as potentially similar to A, we either have a common method (Algorithm 7, line 5), in which case the weight is 1, the method either belongs only to B and is similar to one of the A's methods (Algorithm 7, line 14).

$$\begin{aligned} num &= \sum_{key=B} w = \sum_{\substack{key=B \\ common=\textbf{true}}} w + \sum_{\substack{key=B \\ common=\textbf{false}}} w \\ &= |A^\star \cap B^\star| + \sum_{M \in A^\star \setminus B^\star} \sum_{M' \in B^\star \setminus A^\star} 2 \cdot msim'(M, M') \\ &\geq |A^\star \cap B^\star| + 2 \sum_{M \in A^\star \setminus B^\star} \max_{M' \in B^\star \setminus A^\star} msim'(M, M') \\ &\geq |A^\star \cap B^\star| + 2 \cdot bm(A^\star \setminus B^\star, B^\star \setminus A^\star) \end{aligned}$$

From Lemma 1 we have that the variable s computed at line 9 in Algorithm 8 is greater or equal than $dsim(A, B)$. This means that if $s < \theta_p$, B cannot be similar to A. We will only compute the real deep similarity only if $s \geq \theta_p$.

6 Experimental results

6.1 Algorithms running times

We have created the database described in the previous section for a collection of 1,165,942 Android samples from the Google Play market. The collection comprised mostly of free apps, but we also had a small budget (a couple of hundreds US dollars) in order to purchase the most popular paid apps.

The database was created from an initial collection of 1,065,000 samples that grew as new applications were added. The initial construction took 4 days and 15 h. After that, several thousands new samples were added daily.

Table 1 shows the total running times of the three functions used for the database construction. The MAP-1 function took 76.7 h or 69 % of the total time. From this time, 94 %

Table 1 Running times for database construction

Operation	Running time (h)
MAP-1	76.7
REDUCE-1	27.7
REDUCE-2	6.46

or 72.1 h were spent parsing the binary applications in order to extract the methods and the n-grams. Both REDUCE-1 and REDUCE-2 times also contain the time spent by the framework to build the list of values for each key.

Having the database built, the average running time for finding all pairs of similar applications based on the shallow similarity (Algorithm 5) takes 2 h and 37 min.

Both searches for similar applications with a given one vary with the number of methods of the searched sample. If we use the shallow similarity algorithm, the running time takes from less than a second to a couple of seconds. Deep similarity is more costly, varying from a few seconds for small applications to a few minutes on average and more than 10 min for large applications.

6.2 Similarity cases found by the system

Before discussing the results found by the system, we must point out that the similarity of two application could indicate different things:

- they both may be using the same framework but with a slightly change configuration (e.g., a framework for on-line radio but with different addresses from where the application uses the radio-stream).
- one could be an re-branded version of the other (e.g., the vendor of the first application is willingly selling it's code to another vendor; the other vendor usually only applies some UI changes to the original app (colors, skin, images, icon, texts, etc).
- finally, one could be a copy of the other one, without any previous agreement between the two vendors (plagiarism).

Although our goal is to find only plagiarism cases (the third case), the system will also output some pairs that belong to the first two cases.

The first case should be ruled out by the library code identification that was previously discussed. In practice, a popular framework that is used by enough different developers or by many different applications will be marked as library code automatically. The methods extracted from other frameworks/libraries can be manually set as library code. However, the system may still have some false alarms for different applications that use the same uncommon framework.

Unfortunately the second case is almost impossible to rule out automatically. The difference between a re-branding and a plagiarism case can consist only in some agreement between the vendors that cannot be inferred by an automatic system and sometimes not even by a human.

For the plagiarism cases we will also be interested in the monetization techniques used by the attacker. We have established in the introductory section that 1 h could suffice to create a repackaged app. The 25$ payment is still needed for publication so there must be methods for getting more money out of a repackaged app. The answer lies in the third step from Fig. 1, the modifications performed to the original application. The most common changes that can easily be applied are:

- if an Adware SDK is used, change the ad sdk unique identifier. This identifier tells the server where to put the virtual money that is earned by clicking an add. In many cases this is merely a string that can be easily replaced with another one (for example, the following string represents an add id for Google AdMob SDK: "UA-99999999-9")
- add another Adware SDK preferable one that can offer more money through advertising. This is a little bit more difficult than the first approach as it requires an integration with the application. However, several Ad SDK offers push notification a technique that does not

Table 2 Example of application that was plagiarized

Package	Vendor	Installs	Price
com.vectorunit.red	Vector unit	1,000,000 5,000,000	~2.3 Euro
com.vectorunit.red.bdroid.a12b2f30	Jennifer Nelson	100–500	Free
com.vectorunit.redbbzf50	Laura Stone	1000–5000	Free
com.vectorunit.red.bdroid.a12b2102	Janet Stofkoper	1000–5000	Free
com.vectorunit.red.bdroid.a12b2100	Joshua Parker	1000–5000	Free

require a special integration with the host app, but only a separate thread that from time to time will alert the user about different promotional offers. Furthermore, push notifications do not require the app that is hosting the SDK to be active for them to work. This actually makes this technique more expensive (e.g., having push notification usually means more money for the one that is hosting them) as the user can be spammed continuously with promotional messages.
– remove some components that are not required (remove some ad SDK or different payment methods that the app is using).

Table 2 shows an example of such a case. The first line represents the original application. The next ones are different copies of it. In this case the original app was not free. The copies, however, were free and bundled with the following Adware SDK: AirPush, StartApp. Even if all of the copies had different vendors they all share the same Ad identifier for the two adware SDK; this is a clear indicator that the same entity was behind this action. Furthermore, additional permissions was added to the repackaged forms:

- `android.permission.VIBRATE`
- `android.permission. ACCESS_COARSE_LOCATION`
- `android.permission. ACCESS_FINE_LOCATION`
- `android.permission.ACCESS_WIFI_STATE`
- `android.permission.READ_PHONE_STATE`
- `com.android.launcher.permission. INSTALL_SHORTCUT`
- `com.android.launcher.permission. UNINSTALL_SHORTCUT`

All of these permissions were required by the AdSDK. For example, `INSTALL_SHORTCUT` offers the AdSDK the possibility of adding icons to the main desktop (each added icon gives the vendor some money).

To make it even more convincing, the repackaged app had a similar icon with the original one (actually the mirror image of the original icon). The two icons are depicted in Fig. 4.

These results represent the state of the market on May 2013. The fake applications were removed (they usually managed to stay up to 10–14 days in the market). Also the number of installs and/or price of the original application may have changed during this time. The names used by the repackaged apps for the vendors are obviously fake (most likely were chosen randomly).

It's not clear how much money the ones that created these fake apps gain; however, it's likely that is more than 25$ per app. Furthermore, assuming that the users that installed the fake apps would have bought the original one, then the original vendor would have gain from 10,000 to 50,000 Euros.

Algorithm 5 ran on the entire collection of 1,165,942 Android samples and found 214,818 pairs of applications whose shallow similarity was above $\theta_p = 50\,\%$. The total number of

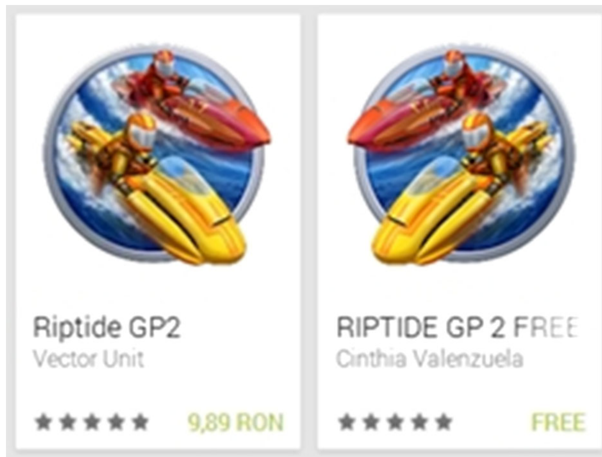

Fig. 4 Example of icon modifications in a repackaged app

Table 3 Statistics on the similar pairs found in the collection

Interval (%)	Nr. pairs	Nr. samples
100	18,709	5587
[90–100)	38,920	14,949
[80–90)	38,182	13,807
[70–80)	32,177	15,486
[60–70)	40,063	17,599
[50–60)	46,767	20,117

unique samples involved was 44,675. The involved samples were also grouped into 4047 clusters using the single linkage approach [28].

We have split the similar pairs by the similarity score as in Table 3. The first category contains the pairs with a perfect match (100 % similarity on non-library methods), while the subsequent categories contain the 10 % length intervals starting from 50 %. The values are also illustrated in the chart from Fig. 5.

For each interval, we have counted the number of pairs with the similarity in that interval, along with the number of unique samples involved. Since an application A can be 95 % similar to an application B and 75 % similar to an application C, the sum on the *Nr. samples* column is greater than the total number of unique samples involved in the pairs.

The involved samples were also analyzed dynamically for behavior that affects the user's privacy. The following undesired actions were found:

- *id* Sending the device unique identifier on the Internet.
- *e-mail* Sending the user's e-mail address.
- *pass* Different user's passwords are sent in plain text.
- *location* Sending the user's GPS location.
- *phone* Sending the phone number.
- *contacts* Sending the contacts from the user's phone agenda.

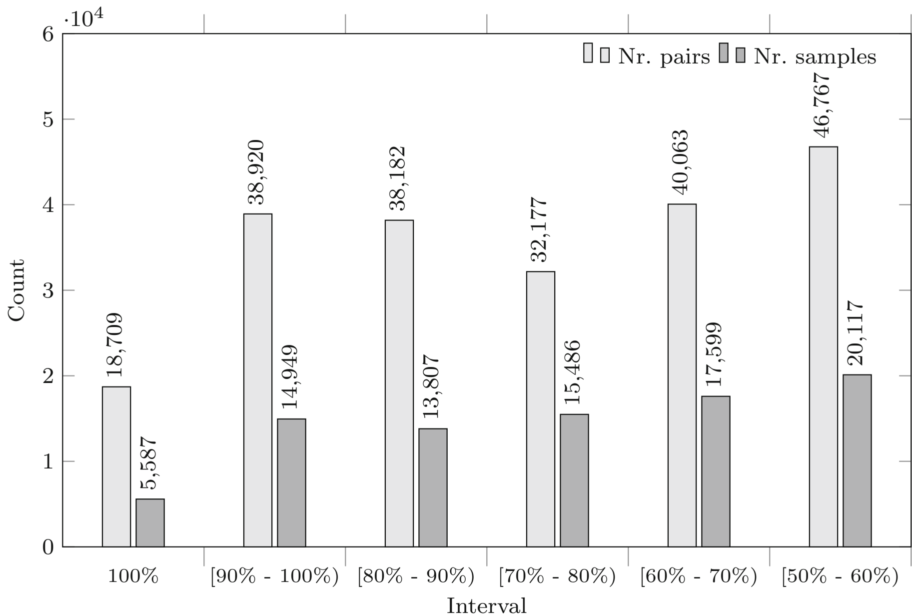

Fig. 5 Statistics on the similar pairs found in the collection

Table 4 Statistics on the similar pairs found in the collection

Interval (%)	*id*	*e-mail*	*pass*	*location*	*phone*	*contacts*
100	777	24	21	74	18	0
[90–100)	2451	157	88	152	69	1
[80–90)	2291	157	85	133	79	1
[70–80)	2702	186	75	171	92	1
[60–70)	3169	192	97	178	103	1
[50–60)	3643	194	102	240	102	1

For each similarity interval, we have counted how many involved applications perform the actions above in Table 4.

Most of the actions from Table 4 are attributed to aggressive Adware SDKs that were introduced by the attacker to gain more financial revenue.

7 Conclusions and future work

We have proposed a new approach for finding plagiarism cases in the Android applications market. Since the number of current applications is over one million, the main concern was scalability.

Two similarity metrics, shallow similarity and deep similarity, can be used to compute how similar two applications are. They are both based on feature extracted from the code, but

the shallow similarity works with methods hashes while the deep similarity also considers similar but non-identical methods based on their OpCode n-grams.

Although deep similarity is slower to compute, the performance is not an issue when checking a single pair of applications. Our solution solves the problem of searching for applications similar to a given one in the entire collection, or even finding all the pairs of similar items.

A database was designed to store all the collection's data in a manner that allows fast retrieval of similar applications. The search is based on the concepts of reversed index and locality-sensitive hashing and is performed using Map-Reduce algorithms. For shallow similarity, we can find all pairs similar to a given one or all similar pairs. For deep similarity, we can perform an approximate search in order to retrieve even heavily obfuscated clones for a given application.

The designed system was able to handle a large collection of 1,165,942 Android samples, from which 44,675 unique ones were involved in 214,818 similarity cases. A dynamic analysis on the involved applications showed that plagiarism not only affects developers but also the users, by performing actions that affect the user's privacy.

As future development, we will continue to work on library code identification, in order to obtain more accurate results.

References

1. Ab Rahman M (2011) Reversing android malware. The Honeynet Project 10th Annual Workshop, Paris, France. https://w.honeynet.org/files/MyCERT-3-PST-HoneynetConf-Reversing%20Android%20Malware.pdf
2. Android Developers (2014) Advertising without compromising user experience. http://developer.android.com/training/monetization/ads-and-ux.html
3. Android Developers (2014) Developer registration. https://support.google.com/googleplay/android-developer/answer/113468?hl=en
4. Android Developers (2014) Signing your applications. http://developer.android.com/tools/publishing/app-signing.html
5. Bloice MD, Wotawa F, Holzinger A (2009) Java's alternatives and the limitations of java when writing cross-platform applications for mobile devices in the medical domain. In: Proceedings of the ITI 2009 31st international conference on information technology interfaces, 2009. ITI'09, pp 47–54. IEEE
6. Burnette E (2009) Hello, android: introducing Google's mobile development platform. Pragmatic Bookshelf. http://dl.acm.org/citation.cfm?id=1816808
7. Chodorow K (2013) MongoDB: the definitive guide. O'Reilly Media, Inc., Sebastopol
8. Crussell J, Gibler C, Chen H (2012) Attack of the clones: detecting cloned applications on android markets. In: Computer security–ESORICS 2012. Springer, Berlin, pp 37–54
9. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
10. Ferrante J, Ottenstein KJ, Warren JD (1987) The program dependence graph and its use in optimization. ACM Trans Program Lang Syst (TOPLAS) 9(3):319–349
11. Gartner (2014) Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013. http://www.gartner.com/newsroom/id/2665715
12. Hanna S, Huang L, Wu E, Li S, Chen C, Song D (2013) Juxtapp: a scalable system for detecting code reuse among android applications. In: Detection of intrusions and malware, and vulnerability assessment. Springer, Berlin, , pp. 62–81
13. Heintze N, et al (1996) Scalable document fingerprinting. In: 1996 USENIX workshop on electronic commerce, vol 3
14. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on theory of computing, pp 604–613. ACM
15. Kuhn HW (1955) The hungarian method for the assignment problem. Naval research logistics quarterly 2(1–2):83–97

16. Lafortune E, et al (2009) Proguard. http://www.guardsquare.com/files/media/slides/ProGuard_IstanbulTechTalks2014.pdf
17. Leurent G (2008) Md4 is not one-way. In: Fast software encryption. Springer, Berlin, pp. 412–428
18. Manber U, et al (1994) Finding similar files in a large file system. In: Proceedings of the USENIX winter 1994 technical conference, vol 1. San Fransisco, CA, USA
19. Octeau D, Jha S, McDaniel P (2012) Retargeting android applications to java bytecode. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, p 6. ACM
20. Oprişa C, Cabău G, Coleşa A (2014) Automatic code features extraction using bio-inspired algorithms. J Comput Virol Hacking Tech 10(3):165–176
21. Oprisa C, Checiches M, Nandrean A (2014) Locality-sensitive hashing optimizations for fast malware clustering. In: 2014 IEEE International conference on intelligent computer communication and processing (ICCP), pp 97–104. IEEE
22. Potharaju R, Newell A, Nita-Rotaru C, Zhang X (2012) Plagiarizing smartphone applications: attack strategies and defense techniques. In: Engineering secure software and systems. Springer, Berlin, pp 106–120
23. Protsenko M, Muller T (2013) Pandora applies non-deterministic obfuscation randomly to android. In: 2013 8th International conference on malicious and unwanted software:“ The Americas”(MALWARE), pp 59–67. IEEE
24. Rajaraman A, Ullman JD (2012) Mining of massive datasets. Cambridge University Press, Cambridge
25. Roy CK, Cordy JR (2007) A survey on software clone detection research. Tech. rep., Technical report 541, Queens University at Kingston
26. Schleimer S, Wilkerson DS, Aiken A (2003) Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD international conference on management of data, pp 76–85. ACM
27. Shabtai A, Moskovitch R, Feher C, Dolev S, Elovici Y (2012) Detecting unknown malicious code by applying classification techniques on opcode patterns. Secur Inform 1(1):1–22
28. Sibson R (1973) Slink: an optimally efficient algorithm for the single-link cluster method. Comput J 16(1):30–34
29. Turing AM (1936) On computable numbers, with an application to the entscheidungsproblem. J Math 58(345–363):5
30. Warren C (2013) Google play hits 1 million apps. http://mashable.com/2013/07/24/google-play-1-million/s
31. Zhou Y, Jiang X (2012) Dissecting android malware: characterization and evolution. In: 2012 IEEE symposium on security and privacy (SP), pp 95–109. IEEE

Ciprian Oprişa received his B.Sc. in Computer Science from Technical University of Cluj-Napoca, Romania, in 2011 and his M.Sc. in Communication Networks and Distributed Computing at the same university, in 2013. He is currently working on his Ph.D. thesis, where he aims to solve security problems through big data and machine learning techniques. He is also a Team Lead in the Antimalware Lab at Bitdefender, where he performs reverse engineering on malicious applications and develops new technologies for heuristic malware detection and for fighting network-based threats. His research interests include security, machine learning and algorithms.

Dragoş Gavriluţ is a Lecturer at Alexandru Ioan Cuza University of Iaşi, where he received his Ph.D. in 2012, with the thesis entitled "Meta-heuristics for Anti-Malware Systems." He received his B.Sc. and M.Sc. in Computer Science from the same university, in 2004 and 2006, respectively. He is also an Antimalware Research Manager at Bitdefender, managing team of 60+ people that develops heuristic detections, cloud-based services, system testing services, disinfection routines, Android and iOS analysis, event correlation algorithms, data mining, IoT and cyber security analysis.

George Cabău received his B.Sc. in Computer Science from Technical University of Cluj-Napoca, Romania, in 2010 and his M.Sc. in Communication Networks and Distributed Computing at the same university, in 2013. He is currently a Ph.D. student, researching big data techniques and building large-scale antimalware systems. He is also a Technical Project Manager at Bitdefender, coordinating the Antimalware team.