Semantics-Based Repackaging Detection for Mobile Apps

Quanlong Guan[1(✉)], Heqing Huang[2], Weiqi Luo[1], and Sencun Zhu[2]

[1] Jinan University, Guangzhou, China
{gql,luoweiqi}@jnu.edu.cn
[2] Department of Computer Science and Engineering,
The Pennsylvania State University, University Park, PA 16802, USA
{hhuang,szhu}@cse.psu.edu

Abstract. While Android app stores keep growing in size and in number, app repackaging has become a major threat to the health of the mobile ecosystem. Different from many syntax-based repackaging detection techniques, in this work we propose a semantic-based approach, RepDetector, which is more robust against code obfuscation attacks. To capture an app's semantics, our approach extracts input-output states of core functions in the app and then compare function and app similarity. We implement a prototype of RepDetector, and evaluate it against various obfuscation technologies. The results show that our approach can detect repackaged apps effectively. It is also at least a hundred times faster than Androguard.

1 Introduction

In recent years, the mobile application world has been expanding dramatically. As of Oct. 2015, Google Play has over 1.5 millions of apps available for downloading. Since high popularity leads to more downloads, many popular Android apps have been copied or repackaged in recent years. Attackers can easily repackage an app under their own names or embed advertisements to earn pecuniary profits. They can also modify a popular app by inserting malicious payloads into the original app and leverage its popularity to accelerate malware propagation. Moreover, because of the popularity of the Android platform, many unofficial app markets exist. Most of them do not perform careful sanity check of uploaded apps. Thus, app repackaging in the Android platform has become very serious, which is increasingly hurting the app ecosystem. App developers are discouraged for loss of revenue, and app users may be deceived from installing malware.

To detect app repackaging, recently various approaches have been explored. Some of them detect repackaged or cloned apps based on code features, such as DroidMOSS [30], Juxtapp [13], DNADroid [8], AnDarwin [12]. For example, DroidMOSS [30] extracts app fingerprints through fuzzy hashing; Juxtapp [13] uses feature hashing for code similarity analysis. While such approaches are capable of recognizing code that is syntactically similar, they are not effective under semantic-preserving obfuscation, where repackaged apps are functionally

J. Caballero et al. (Eds.): ESSoS 2016, LNCS 9639, pp. 89–105, 2016.
DOI: 10.1007/978-3-319-30806-7_6

the same but syntactically different. The syntactic differences include instruction reordering, interleaved methods, opaque branch insertions or the substitution of semantically equivalent control structures. Moreover, new evasion solutions or obfuscation methods have also been explored recently [15,27]. Hence, false negative rates of these approaches could be very high under such attacks.

In this work, we take a semantic-based approach to detecting repackaged apps. In our approach, named RepDetector, the input-output relationship of a function is captured to express its semantics. As long as a repackaged app preserves the critical semantics of the original app, according to our approach, their similarity would be high. In summary, this paper makes the following technical contributions:

- We propose a semantic-based approach to detecting repackaged apps. It can tolerate certain noise insertion or sophisticated obfuscation, and hence is obfuscation-resilient.
- We capture the input-output states of functions with state flow graphs to describe the semantic behaviors of an app. Then we introduce an effective algorithm to compare the similarity of functions from different apps by an SMT solver and detect semantic repackaging with Mahalanobis distance.
- We implement a prototype of RepDetector, and evaluate its detection accuracy and efficiency with both known repackaged apps, obfuscated apps, and apps from Google Play. The result shows that RepDetector is capable of scanning real-world Android repackaging apps with obfuscation resiliency. It is also over a hundred times faster than a well-known detection tool Androguard [4].

2 Overview

Problem Statement: Mobile app repackaging is an approach used by an adversary to change an existing app while keeping its main functionality. After altering an app's code, data, ad library, or structure, the adversary re-publishes the new app to the app store for profits or malware propagation. The cost of repackaging an Android app is low. An app's bytecode can be disassembled into an intermediary representation, which is easy for human to read. After that, the code may be quickly understood and extra code may be inserted. After performing the bytecode manipulation on the intermediary representation, the modified version can be directly assembled back to a functional Android application with tools like APKtool or Smali/Baksmali. Although obfuscation tools like Proguard [18] and Dexguard [3] may be employed to confuse the adversary, obfuscation is not sufficient to prevent repackaging. An experienced adversary may obtain the data/control flow graphs and guess the meaning of functions in the app. Moreover, the adversary may modify the app's code through various obfuscation techniques while keeping its function equivalent. It will be harder to detect this type of repackaging behavior.

Figure 1 shows a running example. We consider the Dalvik bytecode of two methods(*function_add(II) and function_Grid(III)*) from a legitimate app(minion-fun-1.2.apk) and a repackaged app(card-sharks.apk), respectively.

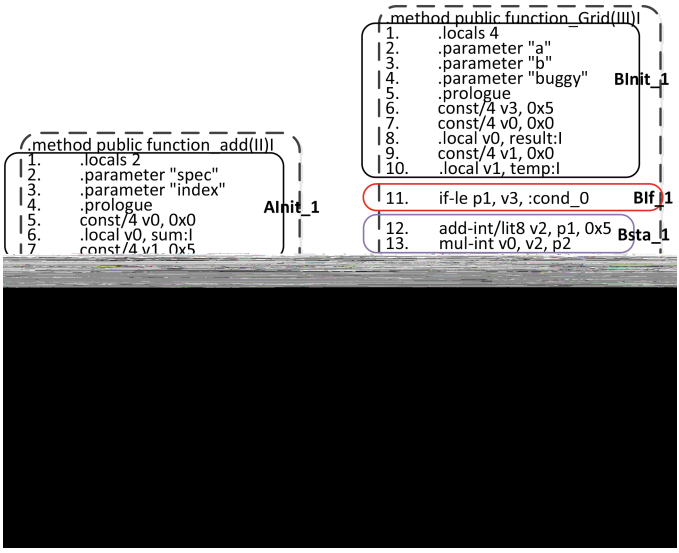


Fig. 1. Dalvik bytecode of an app and its obfuscated version

function_add(II) (*Segment I*, for short) has been divided into several blocks, labeled as {*Ainit_1, Alf_1, Asta_1, Aret, Asta_2*}. Likewise, the *function_Grid (III)* (*Segment II*, for short) is also split into several blocks, labeled as {*Binit_1, Blf_1, Bsta_1, Bret, Bsta_2, Blf_2, Bsta_3, Bsta_4*}. The code between *Segment I* in legitimate app and *Segment II* is quite different. Some noisy variables, such as {*buggy,temp*}, and redundant instructions have been injected in *Segment II*. These instructions contain opaque branch and junk code, labeled as {*Blf_2, Bsta_3 and Bsta_4*}. In fact, the functional behaviors of both methods in Fig. 1 are extremely similar: both perform the similar calculation and return the same result in the end, although *Segment II* has obfuscated the syntax structure of *Segment I*.

The similarity between Android programs can be reflected in different aspects, including syntax-level birthmark, GUI features, and resource features. Syntax features [13,21], such as fingerprints or feature hashing of mobile code, have been applied for repackaging detection. However, such schemes will not work well under semantic-preserving transformation/obfuscation.

Architecture: To detect repackaging under code obfuscation, we propose RepDetector in this work. The architecture of RepDetector is described in Fig. 2. The inputs are the Android apps probably from different Android markets. RepDetector consists of four major modules: core class and function extraction, function output states construction, function similarity measurement and app similarity measurement. Core functions along with important classes will be extracted according to bytecode and the manifest file of an app. We then construct a state flow graph for these core functions and compute the output

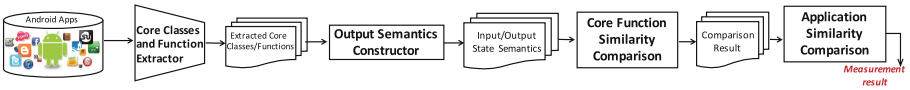


Fig. 2. The architecture of RepDetector

states of each function. With a Satisfiability Modulo Theories (SMT) solver, we then check the semantic equivalence of two core functions based on their output states. Finally, we quantify the similarity between two apps (each consisting of some core functions) using Mahalanobis distance. Our method uses symbolic execution, but it is simplified to compute by merging the flow states. The details of our approach are presented in the next section.

3 System Design of RepDetector

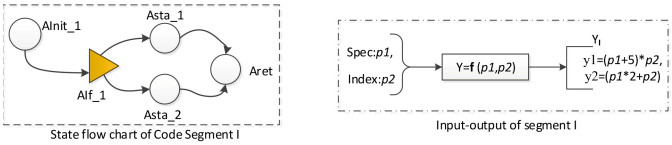
3.1 Core Classes and Functions Extraction

Each APK file contains files like *classes.dex, AndroidManifest.xml*, and subdirectories like *Res, META-INF*. The *classes.dex* file is a Dalvik executable generated from the compiled classes. Tools like *Dadexer* [1] *and Dex2jar* [2] can be used to decompile it into classes directly from the APK or JAR file. We note that not all classes are relevant to the core functionality of an app, while repackaging keeps the app's core functionality. Hence, for repackaging detection, we will only consider the functionality-relevant classes. The manifest file(*AndroidManifest.xml*) presents essential information about an app to the Android system. The functionality-relevant classes include principal components of an app: activities($< activity >$), services ($< service >$), broadcast receivers ($< receiver >$), and content providers ($< provider >$). We directly retrieve a list of such classes by parsing the *AndroidManifest.xml* file. and then construct the Class Invocation Graph (CIG) for the list of classes.

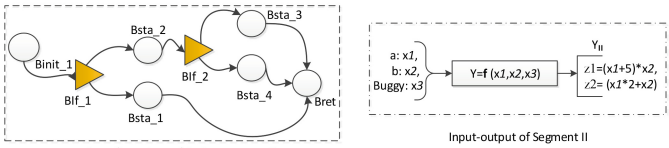
In a CIG, each node represents a class, and a directed edge between two nodes represents the existence of a function (i.e., method) invocation relationship between them. Our CIG takes into account class relationship and function invocations by static analysis. Moreover, we calculate the weight for each class in the CIG based on several attributes (e.g., fan-in and fan-out of a class node in CIG). By setting a threshold on weight, we can filter out the classes whose weights are below the threshold. The remaining classes are called *core classes*. Following a similar procedure, we can identify the *core functions* in these core classes. Another condition for core functions is that they must be created and defined by the developer (or a potential re-packager), not defined by Android libraries or third-party libraries.

Specifically, we create a whitelist, which includes Android SDK and third-party libraries. When analyzing an individual core method, we will check whether it invokes other methods. If an invoked method is from Android SDK or a third-party library in the whitelist, we will summarize its features such as class

type and variable types. With such features, we can determine whether two invoked methods are equivalent or not in a later stage. On the other hand, when an invoked method is also defined by the developer (or a potential repackager), we will follow into the invoked method. This process is repeated until an invoked method contains no user defined method. The output states of a callee method are then jointed into the states of other instructions in the caller method. Our work does not consider function invocations in native code though. It was reported that only a small fraction (5 %) of apps in the Android market contain native codes [30].



(a) The state construction for Segment I.



(b) The state construction for Segment II.

Fig. 3. The state flow chart and input-output state for Segment I and Segment II.

3.2 Output Semantics Construction

Our next step is to analyze the structure of these core functions through State Flow Chart (SFC). An SFC is constructed from control flow graph (CFG) of every core function, and the nodes in the SFC are either basic blocks in the CFG or relevant conditional instructions (e.g., if-branching instruction). Thus, the SFC is a directed graph that clearly displays how the program states flow among *instructions*. Using the code segments in Fig. 1 as an example, the SFCs of these two segments are extracted and shown in Fig. 3. In the left hand of Fig. 3(a), Segment-I includes initial block, one constraint, two statements, and one return. There are five nodes in total: {*Ainit_1, Alf_1, Asta_1, Aret, Asta_2*}. Segment-II has two constraints and eight nodes, as showed in the left hand of Fig. 3(b): {*Binit_1, Blf_1, Bsta_1, Bret, Bsta_2, Blf_2, Bsta_3, Bsta_4*}.

Definition 1 *(Function's Output State). Let set* $P = [p_1, ..., p_n]$ *be input register parameters (totally* n*) of a core function, and set* $Y = [y_1, ..., y_m]$ *be all possible output states (totally* m*) of the function.* $Y = f(P)$ *is a set of symbolic formulas on* P *generated from the state flow chart of this function.*

From the SFC, we generate the output states by symbolic execution. For example, as shown on the right side of Fig. 3, Segment I has two register parameters: *spec:* p_1 and *index:* p_2; thus, the input parameter set $P_I = [p_1, p_2]$. After several blocks are executed, the output state set Y_I contains two elements y_1, y_2: $y_1 = (p_1 + 5) * p_2$ and $y_2 = (p_1 * 2 + p_2)$. Since Segment II was injected one junk parameter *buggy:* x_3, compared to Segment I, its input parameter set is $P_{II} = [x_1, x_2, x_3]$. Furthermore, some noisy code such as opaque branch was inserted in Segment II. Nevertheless, its output state set Y_{II} contains similar symbols as in Segment I.

3.3 Equivalence Measurement of Two Functions

To determine how similar two apps are, we will perform pairwise similarity measurement between their core functions. Given two core functions from a pair of apps, we measure how semantically similar they are by comparing their input/output states. As it is unknown which input parameters of one function correspond to which parameters of the other function, we hence try different permutations of input parameters to check the equivalence. A form definition is given below.

Definition 2 *(Pairwise Equivalence of Input Register Variables). Given two variable sets:* $P_I = [p_I^1, ..., p_I^n]$ *and* $P_{II} = [p_{II}^1, ..., p_{II}^k]$, $n \leq k$. *let* $\lambda(P_{II})$ *be a permutation of the variables in* P_{II}. *A pairwise equivalence for* P_I *and* P_{II} *is defined as:*

$$\bigwedge_{i=1}^{n} [p_I^i = \lambda_i(P_{II})] \longrightarrow P_I = \lambda(P_{II}).$$

where p_I^i *and* $\lambda_i(P_{II})$ *are the i-th variables in* P_I *and* $\lambda(P_{II})$, *respectively.*

For each output state in one core function, we check whether there exists an equivalent output state in the other core function with certain combination of inputs.

Definition 3 *(Output Equivalence of Two Functions). Let* P_I *and* P_{II} *be the input sets,* Y_I *and* Y_{II} *be the sets of output states, respectively; if we have:* $\forall y_1 \in Y_I, \exists y_2 \in Y_{II}, P_I = \lambda(P_{II}), y_1 = f_I(P_I), y_2 = f_{II}(P_{II})$., *then we check:* $P_I = \lambda(P_{II}) \longrightarrow f_I(P_I) = f_{II}(P_{II})$
where $f_I(P_I)$ *and* $f_{II}(P_{II})$ *are the symbolic formulas of* y_1 *and* y_2, *respectively.*

Example. From Fig. 3, we generate all state results representing the input-output relationship from two function segments.

$$\begin{array}{ll} y_1 = (p_1 + 5) * p_2; & z_1 = (x_1 + 5) * x_2 \\ y_2 = p1 * 2 + p_2; & z_2 = x1 * 2 + x_2; \end{array}$$

We then check the input parameters' equivalence by permutation through Definition 2. Comparing in pairwise their equivalence by Definition 3, we find that

$$\begin{array}{l} (p_1 = x_1) \wedge (p_2 = x_2) \longrightarrow (y_1 = z_1) \\ (p_1 = x_1) \wedge (p_2 = x_2) \longrightarrow (y_2 = z_2) \end{array}$$

The detailed comparison procedure is presented in Algorithm 1. The inputs are the core functions F_I and F_{II} from two different apps. The algorithm mainly involves three steps. First, it analyzes the input variables P_I of function F_I and P_{II} of F_{II}, and then constructs the output state sets Y_I and Y_{II} of two functions by the method **Outstate**(F_I) and **Outstate**(F_{II}). After that, it compares in pairwise each permutation of P_{II} with P_I. For each comparison, it uses Satisfiability Modulo Theories (SMT) to check their equivalence . SMT is a decision procedure that can handle various types of arithmetics and other decidable theories, formulas (e.g.,$(a+b < 5) \wedge (a-c > 3 \vee d < 2)$) with propositional variables. It can decide the satisfiability of formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, bit-vectors, etc. Because Dalvik bytecode is register-based, in this work, our tool extracts the input variables as abstract register variables instead of their actual types. Later when we apply an SMT solver (CVC4 [17]) to check whether the input variables of two functions are equivalent or not, we simplify the procedure by treating the input parameter types as integer and boolean. Theoretically, this simplification could introduce false positives, because different input types may be treated as equivalent, In practice, however, two core functions (i.e., their symbolic formulas) are matched only when their logic is identical, or very similar in the case of false match due to our simplification. For functions with relatively complex logic, false function matching could be unlikely. On the other hand, this simplification has two benefits. First, the input to the SMT solver becomes simpler, making it more efficient in computation. Second, it adds a type of obfuscation resilience when an attacker attempts to evade detection by simply changing variable types (e.g., integer to double or to float). Lastly, we note that some function code contains *intents*, a special type of objects in Android. CVC4 cannot directly recognize and handle intents, so we get around the limitation by using array structure instead. When an app's code uses *intents* to pass data between activities, e.g., through "intent.putExtra()" operations, we simulate such operations of intents by the put operations of arrays.

Second, while the input variables' equivalence is satisfied, the two output states y_i and y_j are checked similarly by **Checksat()** to see whether they are semantically equivalent or not. Specifically, for all permutations $\lambda(P_{II})$ and $(y_i, y_j) \in Y_I \times Y_{II}$, it asks the SMT whether the following formula (the negation of Definition 3) is *unsatisfiable*:

$$P_I = \lambda(P_{II}) \wedge y_i \neq y_j.$$

If so, the comparison result of Checksat() $T_{ij} = 0$. Otherwise, we say the two output states y_i and y_j are equivalent, and the comparison result $T_{ij} = 1$. Then we normalize the accumulative results of comparison by cardinality $m = |Y_I|$ and $n = |Y_{II}|$, respectively. In our running example (Fig. 3), both output variables of two functions are equivalent, so the accumulative comparison result is 2. The size of both output state set is 2, so their normalized similarity score is 1. Now assume Y_I has the same two output states, but Y_{II} has one more output state

Algorithm 1. Semantic Equivalence Measurement of Two Core Functions

Input: Two Core functions from different apps: F_I, F_{II}
Output: *Similarity* sf, $sf^\flat$

$sf = 0$, $sf^\flat = 0$
Temp Matrix T
Set P_I : *the input variables of* F_I,
Set P_{II} : *the input variables of* F_{II},
Set $Y_I \leftarrow$ **Outstate**(F_I), $Y_{II} \leftarrow$ **Outstate**(F_{II})
while $\{\lambda(P_{II})\}$ **do**
  if (**Checksat**$(P_I, \lambda(P_{II}) = 0)$ **then**
    $\lambda(P_{II}) \leftarrow$ *next permutation of combination list for* P_{II};
    Continue;
  else
    for each $y_i \in Y_I$ $(i = 1$ *to* $m)$ **do**
      for each $y_j \in Y_{II}$ $(j = 1$ *to* $n)$ **do**
        Assert $p_I = \lambda(P_{II})$
        $T_{ij} \leftarrow$ **Checksat**(y_i, y_j)
      end for
    end for
    $\lambda(P_{II}) \leftarrow$ *next permutation of combination list for* P_{II};
    $sf \leftarrow max(sf, \frac{\sum_{i=1}^{m} T_{ij}}{m})$, $sf^\flat \leftarrow max(sf^\flat, \frac{\sum_{j=1}^{n} T_{ij}}{n})$;
    Break;
  end if
end while
return sf; $sf^\flat$

(i.e., $n = 3$). Their accumulative comparison result is still 2, but the normalized similarity score is different, depending on the roles of comparison. If Y_I is compared against Y_{II}, the denominator for normalization is $|Y_I| = 2$, so the final score is $sf = 1$. If Y_{II} is compared against Y_I, the denominator is $|Y_{II}| = 3$, so the final score is $sf^\flat = 2/3$. If two functions are the same or very similar, both sf and $sf^\flat$ should be close to 1. On the other hand, for two different functions, both scores should be close to 0. Finally, the algorithm returns the two normalized similarity scores sf and $sf^\flat$.

3.4 Similarity Comparison Between Apps

Finally, RepDetector measures the similarity scores of all core functions between two apps. Given app A and app B, which have k and l core functions, respectively, we perform pairwise comparison of core functions and obtain two similarity scores sf and $sf^\flat$ for each pair based on Algorithm 1. Let SF_i (or $SF_i^\flat$) be the vector consisting of many sfs (or $sf^\flat$s) when comparing the i-th core function of app A with each of l core functions of app B, we define two similarity matrices MF and $MF^\natural$ as follows:

$$MF = \begin{bmatrix} SF_1 \\ SF_2 \\ \cdots \\ SF_k \end{bmatrix}, \quad MF^{\natural} = \begin{bmatrix} SF_1^{\flat} \\ SF_2^{\flat} \\ \cdots \\ SF_l^{\flat} \end{bmatrix}$$

MF is the matrix for comparing app A against app B, and $MF^{\natural}$ is the matrix for comparing B against A. If the two apps are similar, these two matrices would contain similar values. To measure the difference between these two matrices, we use the Mahalanobis distance [10], $d(A, B)$, as our metric. It is a dissimilarity measure between app A and app B of the same distribution with the covariance matrix Σ. For the two apps, their Mahalanobis distance can be calculated as follows:

$$d(A, B) = \sqrt{(MF - MF^{\natural})^T \Sigma^{-1} (MF - MF^{\natural})}. \tag{1}$$

Here, the inverse of matrix Σ^{-1} is the inverse covariance matrix of Σ.

$$\Sigma = \mathrm{E}\left[MF \cdot MF^{\natural}\right] - \mathrm{E}\left[MF\right] \mathrm{E}\left[MF^{\natural}\right]) \tag{2}$$

where $\mathrm{E}\left[MF \cdot MF^{\natural}\right], \mathrm{E}\left[MF\right]$ and $\mathrm{E}\left[MF^{\natural}\right]$ are the expected value or mean of the vectors $MF \cdot MF^{\natural}$, MF and $MF^{\natural}$, respectively. The smaller the Mahalanobis distance, the higher similarity of two apps.

4 Performance Evaluation

In this section, we present the evaluation of our tool RepDetector. The implementation of RepDetector consists of 6380 lines of Java code. Our experiments were conducted on a 20-machine cluster. Each machine is equipped with a Core i7 3.2 GHz CPU and 16 GB memory and its operating system is Ubuntu Linux 10.04.

4.1 Study I: Detection Accuracy with Known Samples

The first objective of our evaluation is to evaluate the detection accuracy of RepDetector, in comparison with Androguard [4] and ViewDroid [27]. We will compare it with a few more detection algorithms later on. We show the number of false positives (FP), number of false negatives (FN), and accuracy ACC [11]. ACC is defined as follows.

$$ACC = 1 - \frac{FP + FN}{\Sigma\ (Total\ Population)} \tag{3}$$

We select 1,000 repackaged app samples from a previous dataset [5]). These apps cover different ranges of file sizes and have been verified as repackaged apps. Specifically, our samples contain 183 groups of apps from various categories, such as game, social, and books. Each group consists of the original app and its repackaged version(s). The maximum number of repackaged apps in one group

Table 1. Comparison of detection accuracy among Androguard,ViewDroid and RepDetector

μ	FP			FN			*ACC*		
	A	V	R	A	V	R	A	V	R
0.7	47	31	9	12	13	0	94.1 %	95.6 %	99.1 %
0.75	32	31	6	23	18	0	94.5 %	95.1 %	99.4 %
0.8	27	23	0	46	25	8	92.7 %	95.2 %	99.2 %
0.85	16	21	0	107	36	19	87.7 %	94.3 %	98.1 %

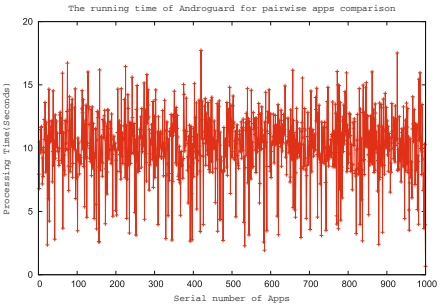
μ: Similarity Threshold. A: Androguard. V: ViewDroid R: RepDetector

is 81. We compare detection accuracy by counting FP (when apps in different groups are reported as repackaging), FN (when apps in the same group are not reported as repackaging) and computing ACC.

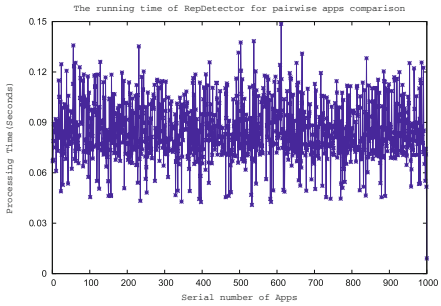
Table 1 shows the evaluation result. In the first column, we set four different similarity thresholds μ, from 0.7 to 0.85. When μ increases, the number of FP instances decreases for all algorithms, but both Androguard and ViewDroid has more FP instances than RepDector. Androguard treats all the bytecode of an app equally, no matter where it comes from; hence, some common libraries could introduce high similarities between different apps. In the case of ViewDroid, some apps in the test dataset are very simple in GUI, so their view graphs extracted by ViewDroid are small with few edges, leading to false positives. RepDetector only examines the core functions and classes, but excludes the third-party libs or the Android framework's bytecode. Thus its FP instances are much less than Androguard. The false negatives of Androguard clearly increases with threshold μ, whereas ViewDroid and RepDector achieve better performance. The FN instances of ViewDroid are mostly caused by some "add-on" functions, which introduce differences in view graphs but with some common functionality code. The last three columns show the detection accuracy (ACC) of these algorithms. The accuracy of RepDetector exceeds 98 %. ViewDroid has stable accuracy at around 95 %. Androguard's accuracy is below 95 % and its lowest accuracy in our experiment setting is 87.7 %, when it reports incorrect results with 123 instances.

4.2 Study II: Efficiency

Figure 4 shows the time for pairwise comparison of apps in our testing app set. Specifically, Fig. 4(a) and (b) depict the average execution time for pairwise comparison by Androguard and RepDetector, respectively. Over 95 % of comparisons in Androguard take 5 $\sim$ 15 s, whereas in RepDetector 98 % pairwise comparisons require less than 0.12 s. From the table in Fig. 4(c), one can further see that the min and max execution times of Androguard are 0.66 s and 17.7 s, respectively. For RepDetector, the min execution time is 0.009 seconds and the max time is 0.14 s. All these numbers indicate that RepDetector is at least one hundred times faster than Androguard. Androguard has low efficiency because



(a) The average time by Androguard



(b) The average time by RepDetector

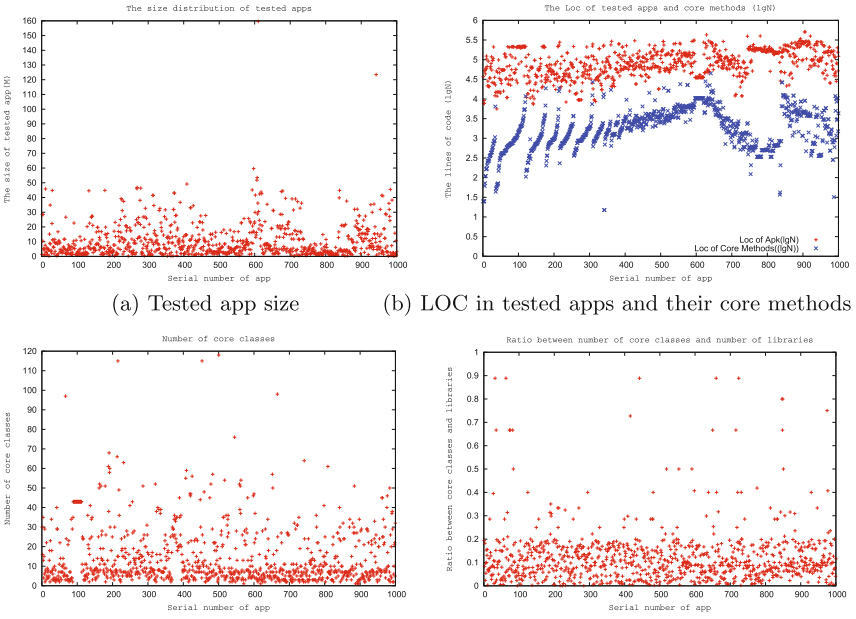
Repackaging Detection Tool	The min Time of app's detection	The Max time of app's detection
Androguard	0.66 s	17.7 s
RepDetector	0.009 s	0.14 s

(c) The min and max time for pairwise comparison of apps by Androguard and RepDetector

Fig. 4. The average running time for pairwise comparison of apps by Androguard and RepDetector

it spends too much time in compressing apps' bytecode by the NCD algorithm and generating features for the apps. Differently, RepDetector only needs to handle core functions and classes, which are only a fraction (sometimes a small fraction) of all functions and classes used by an app. Its handling cost is determined by the efficiency of SMT and the complexity of core functions, not the app size or the total amount of code. We have also compared RepDector with ViewDroid. The average execution time of ViewDroid for testing each pair is about 11 s, much higher than RepDetector. This is because ViewDroid involves relatively expensive subgraph isomorphism detection.

To further explain the high efficiency of RepDetector, we perform a statistical study with our dataset. Figure 5(a) shows the sizes of these 1,000 tested apps. 99 % of them are smaller than 50MB and 80 % are below 20MB. Only two apps are larger than 100MB. Figure 5(b) shows the numbers of lines of code (LOC) in the bytecode of these apps. While 90 % apps have LOC between 10^4 and $10^{5.5}$ ($\lg N$ =4 to 5.5), the LOC in all core methods (selected by setting a weight threshold as 5) are only between $10^{2.5}$ and $10^{3.5}$. This shows a reduction on LOC needed for analysis by around a hundred times, which is the main reason for RepDetector's higher efficiency than Androguard. Figure 5(c) further shows the number of core classes in each app. About 80 % of apps, the number of their core classes is below 30. And 50 % of apps have less than 10 core classes. Finally, Fig. 5(d) shows that in 90 % tested apps, the ratio between number of core classes and number of whitelisted libraries is below 0.2.



(a) Tested app size (b) LOC in tested apps and their core methods (lgN)

(c) Number of core classes (d) Ratio between No. core classes and No. libraries

Fig. 5. The statistics of tested apps

4.3 Study III: Obfuscation Resilience

We now evaluate the robustness of RepDetector against code obfuscation. To obfuscate the original apps for repackaging, we may leverage obfuscation tools like ProGuard [18] and DexGuard [3]), which are able to shrink, optimize, and obfuscate Java source code. However, they cannot perform complicated obfuscation such as data or control flow obfuscation; hence, we resort to another tool [15], which is a dedicated framework for evaluating the obfuscation resilience of Android repackaging detection algorithms. The tool includes 37 types of obfuscation from SandMark [7], including instruction reorder, layout obfuscation, control structure obfuscation and data flow obfuscation, etc. It provides both single obfuscation (each type applies one type of obfuscation) and serialized multiple obfuscation (multiple obfuscation are applied one after another). There are textual or syntactical differences when apps are repackaged by single or serialized multiple obfuscation. For the evaluation, we compare RepDetector with Androguard and three other techniques, including DroidMOSS [30], AST-Distance [21] and AST-Coverage [21]. Except Androguard, the other three tools are not publicly available, so we implemented a version for each of them based on their used algorithms.

We randomly choose 200 apps from our crawled app set and evaluate the true positives and false positives of these algorithms by setting different app similarity detection thresholds. We then use the ROC curve to compare them.

Figure 6 shows the detection results under single or serialized multiple obfuscation. The x-axis is the false positive rate (FPR) and the y-axis is the true positive rate(FPR),

Figure 6(a) shows the ROC curves under single obfuscation. RepDetector performs the best among these algorithms. The detection accuracy of AST-Distance and AST-Coverage drops significantly, while DroidMOSS and Androguard have acceptable performance. As a concrete example, when the false positive rate is 0.32, the true positive rates are: 0.97 (RepDetector), 0.94(Androguard), 0.87 (DroidMoss), 0.72 (AST-Coverage) and 0.63 (AST-Distance), respectively. The ROC curve of AST-Distance is near the diagonal section because AST-Distance can hardly handle obfuscated code.

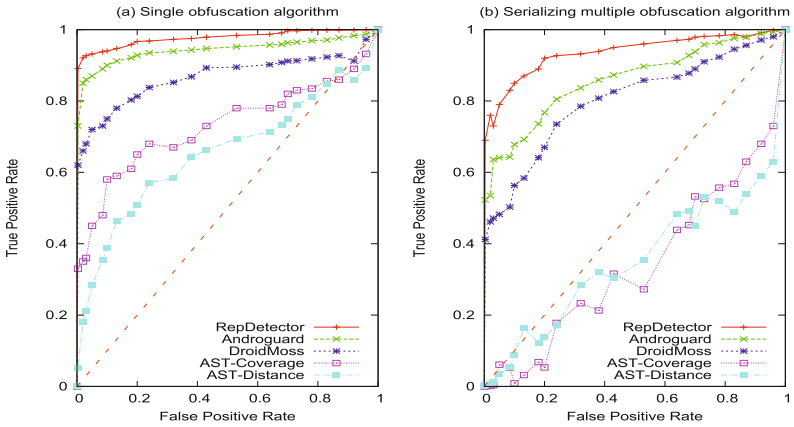


Fig. 6. ROC of algorithm detection accuracy under code obfuscation

We then compare RepDetector with other four detection algorithms under serialized multiple obfuscation. For example, an app may be obfuscated by noise code injection, register rename and followed by opaque branch injection. However, not every app can go through multiple obfuscators successfully. For comparison, we have eliminated the failure cases. Figure 6(b) shows the detection results when apps are obfuscated with duplicate registers, node spliter, buggy code, method madness. We can see that all algorithms degrade their accuracy, compared to the single obfuscation case. This is reasonable because each additional obfuscation changes the code more from its original and makes it harder to detect code similarity. On the other hand, RepDetector still outperforms the other algorithms significantly. Moreover, RepDetector can handle semantics-based obfuscation: computation obfuscation, program reconstruction, data transformation and instruction reordering, because it relies on semantic analysis of each app instead of syntax characteristics. Androguard may partially detect semantic obfuscation such as instruction reordering, but it cannot detect complicated obfuscation. DroidMOSS is less effective than Androguard

and RepDetector because it is mainly based on syntax fingerprints and has hence little effect on code obfuscation. AST-Coverage and AST-Distance have almost no resilience to complicated obfuscation.

5 Discussion

In this section, we discuss some limitations of our work.

Adversaries may split an app's function into multiple smaller functions. In addition, the adversaries may merge several unrelated functions into a big one using redundant code. Crussell et al. [9] mentioned that those kinds of subversions are difficult to detect for most similarity detection methods, including PDG-based ones. That being said, since RepDetector is based on semantics of core functions, the relevant semantics could be merged. Moreover, RepDetector uses Mahalanobis distance to measure the similarity, so it may even tolerate some semantics-changing transforms with an appropriate detection threshold.

Second, RepDetector is limited by the capability of loop analysis. As far as we know, loop analysis is a fundamental challenge for symbolic execution, model checking and other relevant methods. It is hard to determine the actual number of iterations in a loop and even a single loop may generate many different symbolic execution paths when unfolding the loop into a large number of equivalent statements. We also need to specify pre-conditions and post-conditions altogether. In this paper, we make some simplification in handling loops. When the number of iterations cannot be determined, we set an upper bound threshold (in our implementation we set the threshold to 5) and then terminate the iterations. After that many iterations, a sequence of abstract register states will be generated. Although the loop problem could not be perfectly solved by our method, the generated sequence can still catch in some degree the semantic meaning of the loop body. Since our ultimate goal is to detect code similarity, not to understand the exact meaning of a function, we believe our current treatment of this loop problem will not cause big errors.

6 Related Work

Android Application Repackaging Detection. A number of techniques for mobile app repackaging detection have been proposed previously. DroidMOSS [30] computes a series of fuzzy hashes of each method in an app and combines them together. It then compares the fuzzy hashes of two APKs to detect app similarity This approach cannot capture the semantic information of the Dalvik bytecode. PiggyApp [29] uses program dependency graph (PDG) as the core feature to detect piggybacked apps and employ the nearest neighbor searching algorithm to improve scalability. The PDG based approach has also been proposed in DNADroid [8], where the PDG for every method in the Dex file is computed, and a graph isomorphism algorithm is used to calculate the similarity between the computed PDGs. AnDarwin [9] improves DNADroid's detection efficiency by extracting features using LSH and comparing similar apps

with Min-Hash algorithm. PDGs comparison usually suffers from the inefficiency in subgraph isomorphism detection. Also, as shown by Huang et al. [15], it is straightforward for plagiarists to insert redundant code (with data dependency) to obfuscate the PDGs.

In Black-Hat 2011, Androguard [4] was proposed with several techniques on Android app reverse engineering and repackaging detection, It generates the Normalized Compression Distance (NCD [14]) of the signatures extracted from the victim and suspicious apps to determine the similarity. Signatures are created based on features extracted from a generalized representation of the Dalvik bytecode – the opcode sequences. Although this approach is able to capture the high-level semantic information of the code, including control flow graph (CFG), a simple CFG flattening obfuscation can defeat its effectiveness on repackaging detection. Huang et al. [15] proposed a framework for evaluating repackaging detection algorithms of Android apps by generating several obfuscators. Our techniques is robust against most of the obfuscators that preserve the semantics of the app.

Recently, Shao et al. [23] showed that it is possible to detect repackaging clones by app resource features. Zhang et al. [27] have proposed a detection technique called ViewDroid, which constructs core features based on the app UI. ViewDroid is robust against most of the code-level obfuscation. However, for apps with a small set of UI components, it might easily produce some false positives. SmartDroid [28] also uses user interfaces to find user interactions that will trigger sensitive APIs dynamically. Chen et al. [6] proposes a code clone detection technique, which performs the geometry-characteristic-based encoding of control flow graph. While the technique is computationally efficient, it cannot deal with app repackaging using code obfuscation techniques. We note that these techniques (including ours) target at solving different challenges on repackaging detection, they each have some unique strengths. Therefore, they are complementary to each other in nature.

Software Plagiarism Detection. MOSS [22] applies local fingerprinting to detect source code plagiarism. Lim et al. [19] leveraged stack pattern based birthmark, which requires the source code and are vulnerable to some types of code obfuscation. Myles et al. [20] analyzed executables statically and used k-gram to perform similarity measurement. However, it is not robust to instruction reordering and junk instruction insertion. There are also dynamic software birthmarks based software plagiarism detection methods, including core values based birthmark [16,26] and system call based birthmark [24,25]. These dynamic methods are not efficient enough for large-scale repackaging detection in Android markets.

7 Conclusions

App repackaging is a way to change an original app into one with the same or similar functionality. It has become a popular tool for malware propagation and for piracy. It is very important for repackaging detection algorithms to be resilient

to code obfuscation. While many approaches have been proposed, scalability and obfuscation resilience remains a challenge. In this paper, we propose a tool called RepDetector, which is designed to find repackaging apps by semantic similarity. Our evaluation has demonstrated its effectiveness and efficiency. In our future work, we plan to conduct a much larger scale evaluation with apps from different markets.

Acknowledgments. We thank the anonymous reviewers for their valuable comments and Dr. Nick Nikiforakis for shepherding our paper. The work of Guan and Luo was supported by the Science and Technology Planning Project of Guangdong Province, China (2014A040401027, 2012A080102007, 2015A030401043). The work of Huang and Zhu was partially supported by NSF CCF-1320605.

References

1. Dedexer. http://dedexer.sourceforge.net/
2. Dex2jar. https://code.google.com/p/dex2jar/
3. Dexguard. http://www.saikoa.com/dexguard
4. Desnos, A.Z.: Androidguard. https://code.google.com/p/androguard/
5. Chen, K., Liu, P., Zhang, Y.: Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In: Proc. of ICSE (2014)
6. Chen, K., Wang, P., Lee, Y., Wang, X., Zhang, N., Huang, H., Zou, W., Liu, P.: Finding unknown malice in 10 s: mass vetting for new threats at the google-play scale. In: Proceedings of the 24th USENIX Conference on Security Symposium, pp. 659–674. USENIX Association (2015)
7. Collberg, C.S., Myles, G., Huntwork, A.: Sandmark-a tool for software protection research. IEEE Secur. Priv. **1**(4), 40–49 (2003)
8. Crussell, J., Gibler, C., Chen, H.: Attack of the clones: detecting cloned applications on android markets. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 37–54. Springer, Heidelberg (2012)
9. Crussell, J., Gibler, C., Chen, H.: Scalable semantics-based detection of similar android applications. Technical report (2012). ucdavis.edu
10. De Maesschalck, R., Jouan-Rimbaud, D., Massart, D.L.: The mahalanobis distance. Chemom. Intell. Lab. Syst. **50**(1), 1–18 (2000)
11. Fawcett, T.: An introduction to ROC analysis. Pattern Recogn. Lett. **27**(8), 861–874 (2006)
12. Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., Choi, H.: Characterizing android application plagiarism and its impact on developers. In: Proceedings of MobiSys (2013)
13. Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., Song, D.: Juxtapp: A scalable system for detecting code reuse among android applications. In: Proceedings of DIMVA (2013)
14. Hemel, A., Kalleberg, K.T., Vermaas, R., Dolstra, E.: Finding software license violations through binary code clone detection. In: Proceedings of MSR. ACM (2011)
15. Huang, H., Zhu, S., Liu, P., Wu, D.: A framework for evaluating mobile app repackaging detection algorithms. In: Huth, M., Asokan, N., Čapkun, S., Flechais, I., Coles-Kemp, L. (eds.) TRUST 2013. LNCS, vol. 7904, pp. 169–186. Springer, Heidelberg (2013)

16. Jhi, Y.C., Wang, X., Jia, X., Zhu, S., Liu, P., Wu, D.: Value-based program characterization and its application to software plagiarism detection. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 756–765. ACM (2011)
17. King, T., Barrett, C., Tinelli, C.: Leveraging linear and mixed integer programming for SMT. In: Formal Methods in Computer-Aided Design, FMCAD 2014, pp. 139–146. IEEE (2014)
18. Lafortune, E.: Proguard. http://proguard.sourceforge.net/
19. Lim, H., Park, H., Choi, S., Han, T.: Detecting theft of Java applications via a static birthmark based on weighted stack patterns. IEICE - Trans. Inf. Syst. **E91–D**(9), 2323–2332 (2008)
20. Myles, G., Collberg, C.S.: K-gram based software birthmarks. In: SAC (2005)
21. Potharaju, R., Newell, A., Nita-Rotaru, C., Zhang, X.: Plagiarizing smartphone applications: attack strategies and defense techniques. In: Proceedings of ESoSS (2012)
22. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In: Proceedings of ACM SIGMOD International Conference on Management of Data (2003)
23. Shao, Y., Luo, X., Qian, C., Zhu, P., Zhang, L.: Towards a scalable resource-driven approach for detecting repackaged android applications. In: Proceedings of ACSAC. ACM (2014)
24. Wang, X., Jhi, Y., Zhu, S., Liu, P.: Behavior based software theft detection. In: Proceedings of 16th ACM Conference on Computer and Communications Security (CCS) (2009)
25. Wang, X., Jhi, Y.C., Zhu, S., Liu, P.: Detecting software theft via system call based birthmarks. In: Computer Security Applications Conference, ACSAC 2009. Annual, pp. 149–158. IEEE (2009)
26. Zhang, F., Jhi, Y., Wu, D., Liu, P., Zhu, S.: A first step towards algorithm plagiarism detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM (2012)
27. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of ACM WiSec, pp. 25–36. ACM, New York, NY, USA (2014)
28. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., Zou, W.: SmartDroid: an automatic system for revealing UI-based trigger conditions in Android applications. In: Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, pp. 93–104. ACM (2012)
29. Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of piggybacked mobile applications. In: Proceedings of ACM CODASpPY (2013)
30. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of ACM CODASpPY (2012)