Autom Softw Eng
DOI 10.1007/s10515-015-0182-6

Detecting plagiarized mobile apps using API birthmarks

Daeyoung Kim[1] · Amruta Gokhale[1] · Vinod Ganapathy[1] · Abhinav Srivastava[2]

Received: 9 December 2014 / Accepted: 26 June 2015

Abstract This paper addresses the problem of detecting plagiarized mobile apps. Plagiarism is the practice of building mobile apps by reusing code from other apps without the consent of the corresponding app developers. Recent studies on third-party app markets have suggested that plagiarized apps are an important vehicle for malware delivery on mobile phones. Malware authors repackage official versions of apps with malicious functionality, and distribute them for free via these third-party app markets. An effective technique to detect app plagiarism can therefore help identify malicious apps. Code plagiarism has long been a problem and a number of code similarity detectors have been developed over the years to detect plagiarism. In this paper we show that obfuscation techniques can be used to easily defeat similarity detectors that rely solely on statically scanning the code of an app. We propose a dynamic technique to detect plagiarized apps that works by observing the interaction of an app with the underlying mobile platform via its API invocations. We propose *API birthmarks* to characterize unique app behaviors, and develop a robust plagiarism detection tool using API birthmarks.

Keywords Mobile apps · Plagiarism · API birthmarks

1 Introduction

In recent years, smart phone app markets have witnessed explosive growth. Popular app markets, such as those of Apple and Google, now have in excess of a million apps.

✉ Vinod Ganapathy
vinodg@cs.rutgers.edu

1 Rutgers University, Piscataway, NJ, USA

2 AT&T Labs–Research, Bedminster, NJ, USA

With such large numbers, and an equally diverse and large developer community, malpractices in app development should come as no surprise. We focus on *app plagiarism*, the practice of using another developer's code, without permission or payment, to build and deploy mobile apps.

Plagiarism has long been a problem in traditional software development. It especially affects developers who wish to protect intellectual property embedded in their software. In response, the community has been actively researching techniques to detect plagiarism (Baker 1995; Kontogiannis et al. 1995; Baxter et al. 1998; Krinke 2001; Kamiya et al. 2002; Ducasse et al. 2006; Liu et al. 2006). In the mobile app space, plagiarism is of particular concern. Recent work has shown that plagiarized apps divert advertising revenue as well as the user base from the developers of the original apps (Gibler et al. 2013). More importantly, most mobile malware are repackaged (i.e., plagiarized) versions of otherwise benign mobile apps (Zhou and Jiang 2012). Malicious mobile apps are often distributed via *third-party app stores*, which lure victims by promising free versions of apps that would otherwise require payment on official app stores. The developers of these mobile apps typically obtain a copy of the original app from the official app store, modify the app with their own malicious functionality, and repackage and distribute it via these third-party app stores. The malicious functionality may include displaying unwanted advertisements to victims, monitoring the activities of victims, or even exfiltrating sensitive data to malicious web sites (Felt et al. 2011; Zhou and Jiang 2012; Zhou et al. 2013).

The rise of mobile malware has also spurred the development of anti-malware tools, a variety of which are available through official app markets. These tools use a number of techniques to detect malware, the most common of which is to use simple signature-based scanning to detect malicious apps. Signatures encode code or data patterns seen in malicious apps but not in benign ones. Anti-malware tools scan a new app upon download, and attempt to match them against their signature database. However, it has long been known in the security community that signature-based malware detection tools can easily be evaded using simple transformations (Christodorescu and Jha 2004). Such transformations include variable and function renaming and code reordering, which alter the syntactic structure of the code, so that it no longer matches the signatures used by anti-malware tools. Indeed, recent work has shown that most commercially-available tools to detect mobile malware can easily be evaded using simple transformations (Rastogi et al. 2013).

In response, there have been a number of efforts to develop techniques that detect malicious mobile apps even in the presence of transformations, focusing primarily on detecting plagiarized mobile apps (Zhou et al. 2012, 2013; Crussell et al. 2012; Hanna et al. 2012. To date, these techniques have used static analysis and follow the same basic recipe: Obtain and disassemble a suspect app, and use sophisticated similarily detection algorithms to detect plagiarism. The similarity detection algorithms studied in the literature have primarily been syntactic in nature, ranging from detecting code clones (Hanna et al. 2012), using fuzzy hashing to detect similar code fragments (Zhou et al. 2012), and using machine learning techniques to detect apps that share similar syntactic features (Zhou et al. 2013).

In this paper, we take the position that such static approaches towards detecting plagiarism that rely on syntactic features fundamentally fall short, and can easily be defeated using code obfuscation. We argue that a more robust app plagiarism detection technique is needed, and present such an approach. Our main contributions are two-fold:

(1) *Obfuscation defeats syntactic similarity detection* We show that the use of simple encryption techniques defeats previously-proposed similarity detection tools. These tools rely on syntactic features of the code, and the use of encryption obfuscates these features, making it possible to easily evade detection by these tools. Section 2 presents the results of this study, in which we also study the effectiveness of several off-the-shelf obfuscation tools.

(2) *API birthmarks provide robust plagiarism detection* We develop a robust approach for detecting mobile app similarity. Our approach was inspired by prior work (Schuler et al. 2007) to create unique fingerprints of desktop programs. This approach works by observing the execution of a mobile app—it is therefore dynamic in nature—and recording its interactions with the underlying mobile platform via the API that it exposes. The key idea is that an app can affect the mobile device only by interacting with the platform via its API. Thus, similar apps must interact with the platform in similar ways. We capture "similarity" via the notion of *API birthmarks*, which are subsequences of API calls that are unique to a particular app. In our experiments, we show that the API birthmarks of plagiarized apps substantially resemble those of the original apps. Further, API birthmarks are robust to code obfuscation and encryption because a running app *must* issue API calls to interact with the platform, and these calls themselves cannot be obfuscated. Thus, our approach provides a robust way to detect app plagiarism.

2 Obfuscating mobile apps

Obfuscating a mobile app is the process of transforming the original source code of the app so that resulting code is hard for humans to interpret or read. App developers obfuscate code for a number of reasons, such as to prevent easy interpretion of the reverse-engineered code, or to protect the code from tampering by others. However, obfuscation can also be used to disguise plagiarism, e.g., when a copyrighted app's source code is being reused without obtaining the appropriate permissions. By obfuscating the plagiarized app, the attacker reduces the probability of the new app being identified as similar to the original app. As we will demonstrate, obfuscated apps can easily evade a number of off-the-shelf code similarity detection tools.

2.1 Comparison of obfuscators

There are a number of commercial or free off-the-shelf code obfuscators. Some of these obfuscators target only the traditional desktop software, e.g., DashO and Allatori, while obfuscators such as ProGuard have been ported to work for Android apps as well. These obfuscators use a number of algorithms to transform the original code:

- *Name obfuscation* This technique subtitutes randomly-chosen character sequences in place of the original, human-readable names for a number of code artifacts. For

Table 1 Comparison of different obfuscators in terms of their transformation capabilities

Obfuscators	Transformations				
	Renaming	Dead code removal	Control flow obfuscation	String encryption	Code encryption
ProGuard	✓	✓	×	×	×
Allatori	✓	✓	✓	✓	×
DashO	✓	✓	✓	✓	×
Androcrypt	×	×	×	×	✓

The last row lists the transformations employed by Androcrypt, the obfuscator that we have built

example, the obfuscation may transform source code file names, line numbers, field names, method names, argument names and variable names.

- *Control flow obfuscation* This technique changes the code of an app (either source code or bytecode) to obscure the control-flow structure of the original app. For example, the transformation may target selection and looping constructs and replace them with *goto* statements. When such transformed code is analyzed using a decompiler, it produces code that is difficult to read and understand. Direct jumps may be replaced with indirect ones, which further complicates even basic decompilation tasks, such as constructing a control-flow graph of the program.
- *String encryption* In this technique, string literals in the code are encrypted and code to decrypt these strings is added to the source code. Examples of such string literals include the text of error and exception messages, and the text of the labels or other GUI components such as dialog boxes, buttons, and drop-down menus.

Table 1 shows the transformation techniques used by three off-the-shelf obfuscation tools that can be applied to Android apps. Given the capabilities of these obfuscation tools, we wanted to evaluate their effectiveness. One way to evaluate obfuscators is to feed the original code and obfuscated code to a code similarity detection tool. Code similarity detection tool would report low similarity for strongly obfuscated code, whereas similarity would be high for weakly obfuscated code. Below we describe our experiment.

We randomly chose ten Android apps from a repository of open-source Android apps. The randomly selected apps and the sizes of the corresponding *.apk* files are shown in the Table 2. We then applied ProGuard, Allatori and DashO to each of these apps so as to obtain their obfuscated versions. To compare the original app with its obfuscated counterpart, we used two different code similarity detectors: a popular off-the-shelf tool called Androguard and a state-of-the-art code similarity detector, Juxtapp (Hanna et al. 2012), that was developed in an academic setting.

We observed that the similarity scores reported by both the tools were almost the same. For brevity, Table 3 presents the results of this experiment for Androguard (see also the Appendix for experiments with more apps). Androguard reports a number between 0 and 100 to report similarity: a pair of similar apps will receive a score of 100, and a pair of apps with no similarity will get a score of 0. We converted these numbers suitably to a range between 0 and 1. As Table 3 shows, off-the-shelf obfuscators are somewhat effective at defeating Androguard's similarity detection

Table 2 Size statistics of apps chosen for the experiment to compute similarity measures shown in Table 3

App's name	Size of .apk file
Zxing	720 kB
Connectbot	858 kB
Stardroid	2188 kB
OpenSudoku	211 kB
Pedometer	46 kB
Reddit	759 kB
Amazed	15 kB
Wikinotes	119 kB
Photostream	134 kB
Mileage	366 kB

The first column shows name of the app and second column gives size of the .apk file

Table 3 Results of similarity measure (between 0 and 1) reported by a popular similarity detection tool, Androguard

App name	ProGuard	Allatori	DashO	Androcrypt
Zxing	0.61	0.93	0.56	0
Connectbot	0.60	0.76	0.57	0
Stardroid	0.53	0.82	0.54	0.51
OpenSudoku	0.83	0.82	0.59	0.01
Pedometer	0.84	0.67	0.52	0.03
Reddit	0.47	0.94	0.37	0
Amazed	0.44	0.89	0.75	0.1
Wikinotes	0.92	0.86	0.67	0.28
Photostream	0.77	0.92	0.64	0.03
Mileage	0.72	0.85	0.56	0.56

The Android app's binary executable file. Each column corresponds to app pairs in which obfuscated apps have been obtained by running the corresponding obfuscator

algorithm. Among the three obfuscators and ten apps that we tested, we found that Allatori was the least effective at obfuscating apps. We repeated the same experiment on a larger dataset of 50 apps downloaded randomly from the same repository and found similar results, as shown in the Appendix.

2.2 Androcrypt: an encrypting obfuscator for Android apps

Given the results with the three off-the-shelf obfuscation tools, we asked whether it was possible to build an obfuscator that would be even more effective at transforming an app, so that tools such as Androguard would be rendered ineffective. Drawing on the ideas used to create polymorphic and metamorphic malware (You and Yim 2010), we built *Androcrypt*, an encrypting obfuscator for Android apps. Androcrypt takes a

.apk file corresponding to an Android app as input, encrypts the app and packages it as the payload for a new, obfuscated app.

In more detail, Androcrypt operates as follows. Every Android project consists of a collection of files and directories, in which source code files, binary files and resource files are organized into directories such as *src*, *bin*, *res*, *assets* etc. Typically, raw data files are stored in the *assets* directory which then can be read as a byte-stream using the *android.content.res.AssetManager* class in Android. *AssetManager* class provides lower-level API to open and read the raw files. When supplied an input *.apk* file as input, Androcrypt first creates an empty Android project with all the relevant directories. It then uses the Java cryptography library (we used the *AES/CBC/PKCS5Padding* mode) to encrypt the input *.apk* file, and places it in the *assests* directory. Androcrypt incorporates a new *Activity* class in the new Android project that first reads the encrypted app stored in *assets* directory using *AssetManager* API and decrypts it using the Java cryptography library. The *Activity* then dynamically loads the classes from the DEX bytecode of the decrypted app using the *dalvik.system.DexClassLoader* class. Androcrypt replaces all the original source code files of the application that reside in *src* directory with the single source file of the above *Activity* class. This Android app is then packaged as an *.apk* file and distributed as the obfuscated app. When the app is started, the first component that executes is the new *Activity* class, which decrypts the original app and loads its classes.

While the steps described above suffice to obfuscate a majority of Android apps, there are a few categories of apps that fail to start up when obfuscated this way. Such apps contain classes inherited from either one of two Android classes: *android.content.ContentProvider* or *android.app.Application* In Android, the *ContentProvider* class manages the sharing of data between multiple apps, and is used by apps that share data with other apps. For example, a texting app might use this class if it shares data with the contacts list on the phone. Likewise, the *Application* class is used by Android apps to store any global application state. An Android application using any one of these two classes declares its use in its manifest file. When the Android runtime system executes an Android app, it scans the manifest to determine whether these classes are being used by the app, and first loads these classes before launching the main *Activity* class.

Androcrypt packages the original app and includes the decryption functionality in the main *Activity* class of the repackaged app, which must start first. As a result, the obfuscator cannot extract the individual *ContentProvider* or *Application* classes, thereby breaking the app's functionality. For such apps, Androcrypt uses a hybrid strategy: it obfuscates the *ContentProvider* and *Application* classes using ProGuard, while using encryption on the rest of the *.apk* file.

The last column of Table 3 shows the results of obfuscating apps using Androcrypt. We observe that majority of the apps had low similarity scores when Androcrypt was used as compared to any of the three obfuscation tools. Two exceptions were *Stardroid* and *Mileage* apps. These two apps used classes inherited from the *ContentProvider* and *Application* classes; these classes were not encrypted, which explains why these apps had a higher Androguard-similarity score than the other apps. Thus, Androcrypt's encryption-based approach to obfuscating Android apps is more effective than the techniques used by ProGuard, Allatori and DashO.

Table 4 Mean and median of the similarity measures (between 0 and 1) reported by Androguard for the dataset of 53 apps

	ProGuard	Androcrypt
Mean	0.68	0.07
Median	0.72	0.02

Each column corresponds to app pairs in which obfuscated apps have been obtained by running the corresponding obfuscator

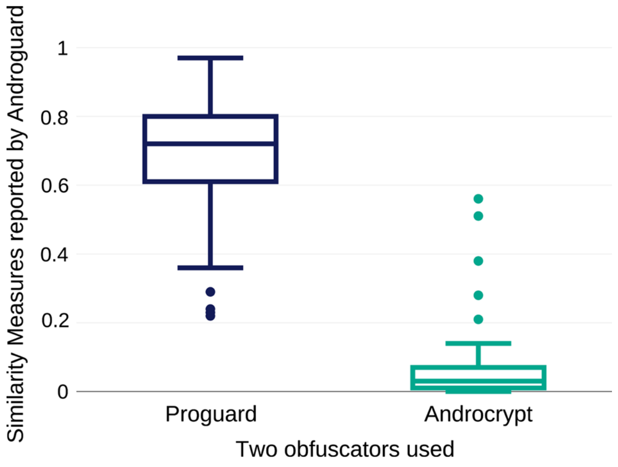

Fig. 1 Distribution of similarity measures reported by Androguard when two different obfuscators were used. *Box on left* corresponds to using ProGuard as the obfuscator whereas *box on right* corresponds to use of Androcrypt as obfuscator. The values used to draw the plots are taken from the columns labeled as (PG, AG) and (AC, AG) in Appendix

We repeated the same experiment on a larger dataset of 53 apps. The complete results are given in Appendix. For brevity, we report the mean and the median of the scores in Table 4, and the box plots of the scores is shown in Fig. 1. As is evident from the plot, the similarity measures reported by Androguard on app pairs in which Androcrypt was used as the obfuscator dropped significantly as compared to the values reported on the same app pairs when off-the-shelf obfuscator, ProGuard, was used. The results demonstrate that simple encryption as used in Androcrypt can easily defeat existing similarity detectors.

3 Methodology

3.1 API birthmark

There are many off-the-shelf program transformation tools available that can modify the source code of the program without affecting the program's functionality. As discussed in Sect. 2, obfuscators are capable of transforming the program so as to evade a popular similarity detection tool for Android.

More generally, a large majority of existing similarity detection tools that have been proposed in the research literature (for Android) rely on simple static properties of the program. Such static properties can be, for example, a unique hash of the executable binary code (Zhou et al. 2012), a feature vector comprising the set of permissions requested by the application and the set of API methods present in the source code of the application (Zhou et al. 2013). An encrypting obfuscator such as Androcrypt completely transforms the syntactic structure of the application, so that any attempt to recover such static properties can be defeated by suitably encrypting the files of the application. Such transformed programs can easily be passed off as the originals, thereby allowing app plagiarism.

In this paper, we develop a robust approach that detects plagiarized mobile apps. Our approach was inspired by an effective technique to uniquely identify desktop programs by creating their *software birthmarks* (Tamada et al. 2004) dynamically. Such a birthmark represents a unique fingerprint of the program that characterizes its runtime behavior. Two programs that have the same birthmark are likely to implement similar functionality, and are likely to have originated from the same source code.

In this approach, birthmarks are *dynamic in nature* and are computed by observing the runtime behavior of the program. As long as there are no significant high-level changes in the behavior of the original and the obfuscated program, the dynamic birthmark of both versions will be similar. Thus, dynamic birthmarks are more robust to program transformation attacks and are more likely to be preserved during obfuscating transformations.

Since a dynamic birthmark is determined by the runtime behavior of the program, it is important that the birthmark captures program properties that constitute the core functionality of the program. If the program's functionality changes, the birthmark of the resulting program must be distinct from that of the original version. However, minor changes in the functionality of the program, as would be expected when an attacker adds new functionality to a plagiarized program, must not cause large deviations in the value of the birthmark.

A desktop program interacts with the environment in which it is run (i.e., operating system) to meet the desired functionality. Similarly, an Android app must interact with the Android and Java runtimes in order to achieve functional goals. These interactions are in the form of the Android and Java API methods invoked by the app. We log these method invocations and collect them in a trace when the app is executed. We leverage prior work (Schuler et al. 2007) to define an API birthmark of the app over the sequence of method invocations by the app. Any new functionality introduced into the app (or old functionality deleted from the original) will likely introduce changes into the sequences of API methods invoked by the app and hence will result in a different birthmark (Table 5).

3.2 API birthmark algorithm

We now describe in more detail the birthmark-based approach for plagiarism detection. Our approach works on a pair of Android apps, and uses birthmarks to compute a similarity coefficient between 0 and 1 (as was the case with Androguard). As explained

Table 5 Snippets from traces of executions of unmodified Android app and its obfuscated version

Trace$_A$	Trace$_{A_{obfs}}$
`Activity.onCreate()`	`Activity.onCreate()`
`Activity.setContentView()`	`Activity.setContentView()`
`Activity.findViewById()`	`Activity.findViewById()`
`View.setVisibility()`	`View.setVisibility()`
`View.setOnClickListener()`	`View.setOnClickListener()`
	`Activity.getWindow()`

Each trace snippet shows the method invoked, and the class in which the method is implemented. Android class `Activity` is prefixed with `android/app` and the class `View` is prefixed with `android/view`. For brevity, we are using abbreviated class names

in Sect. 2, we use Jaccard index to calculate the similarity coefficient. A value closer to 1 indicates high similarity in the observed behavior of apps where as a value closer to 0 signals low similarity in the observed behavior. Although we describe the approach for Android apps, we hypothesize that it will be applicable to other mobile platforms as well. In the description below, A is an unmodified Android app, and A_{obfs} is an app that we suspect is an obfuscated variant of A.

3.2.1 Trace collection

The first step is to run both the apps A and A_{obfs} independently, exercising as much functionality as possible, and collect the execution traces. We then filter the trace so as to retain only those method calls that are invoked on objects whose classes belong to the Android API. We do this filtering because we are interested in observing only the interaction between the app and the underlying Android API. Table 5 shows snippets Trace$_A$ and Trace$_{A_{obfs}}$ of two traces obtained by running A and A_{obfs} respectively, by using the Monkey tool as described in Sect. 5.10 and filtering the corresponding traces. We will use these snippets in our explanation below, although our approach works on the entire trace.

3.2.2 Computing the similarity coefficient

In this step, we compute whether the two apps A and A_{obfs} are similar, using the two traces Trace$_A$ and Trace$_{A_{obfs}}$. This step can be broken down into four sub-steps.

(1) *Collecting object-level API method calls* In an object-oriented language such as Java, a trace of a program is a global sequence of API method calls invoked on objects in the program. We split this global sequence into different *object-level* sub-sequences. Each one of such sequences contains all the API method calls that were invoked on a single object. Such a division avoids to a certain extent the effect of reordering of method calls between different traces introduced by changes in thread scheduling in multi-threaded programs (Schuler et al. 2007).

Let us now turn to the sample trace snippets Trace_A and $\mathsf{Trace}_{A_{obfs}}$ to compute object level API method calls. In these two snippets, we will assume that there is only one object of each of the class types `Activity` and `View`. In general, there could be multiple objects of the same class type, and the sequence of function calls will be collected for each of them individually, based on the object IDs. Let's use the class name itself as the object ID for further simplicity. We accumulate the sequence of function calls invoked on each of the two objects individually. Thus, we get two method sequences each from each of the two traces Trace_A and $\mathsf{Trace}_{A_{obfs}}$. Each of the method calls is prefixed with the full class name such as `android/app/Activity` and `android/view/View` in our actual calculation. We are omitting the class name prefix here for brevity.

API method calls grouped together by objects in Trace_A
`Activity.onCreate();`
`Activity.setContentView();`
`Activity.findViewById();`
`View.setVisibility();`
`View.setOnClickListener()`

API method calls grouped together by objects in $\mathsf{Trace}_{A_{obfs}}$
`Activity.onCreate();`
`Activity.setContentView();`
`Activity.findViewById();`
`Activity.getWindow()`
`View.setVisibility();`
`View.setOnClickListener()`

(2) *Generating k-length sequences* The object-level API method call sequences are long and hence are not easy to compare between different program runs. Schuler et al. (2007) proposed the idea of chopping up these sequences using a sliding window to generate a set of smaller method sequences, and we use this idea as well. Let us assume that for the sample trace snippets Trace_A and $\mathsf{Trace}_{A_{obfs}}$, the length of sliding window is 2. So we can break up the object-level sequences obtained in the above step into a set of sub-sequences, each of length 2 as shown in the table below. From here on, we are omitting the class names which we use in our actual computation.

2-length sequences from Trace_A
`onCreate();setContentView()`
`setContentView();findViewById()`
`setVisibility();setOnClickListener()`

2-length sequences from $\mathsf{Trace}_{A_{obfs}}$
`onCreate();setContentView()`
`setContentView();findViewById()`
`findViewById();getWindow()`
`setVisibility();setOnClickListener()`

(3) *Calculating birthmarks* Finally, we compute birthmark as the union of all 2-length sequences of all objects. The following table shows the birthmarks computed for the two apps A and A_{obfs}. In this case, the union operation merely combines the two sets of sequences without any deletions. But in general, the union will result in deletions of common sequences present among the different sets.

Birthmark B_A for the app A
`onCreate();setContentView()`
`setContentView();findViewById()`
`setVisibility();setOnClickListener()`

Birthmark $B_{A_{obfs}}$ for the app A_{obfs}
`onCreate();setContentView()`
`setContentView();findViewById()`
`findViewById();getWindow()`
`setVisibility();setOnClickListener()`

(4) *Computing similarity coefficient* We use Jaccard index as a measure of similarity between two apps. Given two birthmarks B_A and $B_{A_{obfs}}$ of two apps A and A_{obfs}, the similarity coefficient between two apps is therefore given by Jaccard index of the two sets B_A and $B_{A_{obfs}}$.

$$Sim(A, A_{obfs}) = \frac{\left|B_A \cap B_{A_{obfs}}\right|}{\left|B_A \cup B_{A_{obfs}}\right|}$$

Thus, for our running example of the two apps, the similarity coefficient is equal to:

$$Sim(A, A_{obfs}) = \frac{3}{4} = 0.75.$$

Once we have computed the similarity coefficient between two apps, we use a threshold to decide whether the two given apps should be categorized as plagiarized. Explanation of how a threshold is chosen is given in Sect. 5.3.

```
Input: A trace T consisting of a sequence of ($thread\text{-}id$, $method\text{-}signature$, $object\text{-}id$) entries
Output: Birthmark $B$ containing a set of sequences of function calls
Assign window-length = 4
$L_{obj}$ = list of sequences, each of length window-length, of function calls invoked on a particular object $obj$
$C_{obj}$ = current sequence, whose length <= window-length, of function calls invoked on a particular object obj
$len_{C_{obj}}$ = length of $C_{obj}$
foreach entry ($thread\text{-}id$, $method\text{-}signature$, $obj$) in the trace do
    Add $obj$ to the set of unique object-ids $uniq\text{-}objs$
foreach $obj$ in $uniq\text{-}objs$ do
    Initialize the list $L_{obj}$ to empty
    Initialize current sequence $C_{obj}$ to empty
    Initialize current sequence's length $len_{C_{obj}}$ to be zero
foreach entry ($thread\text{-}id$, $method\text{-}signature$, $obj$) in the trace do
    Append $method\text{-}signature$ to $C_{obj}$
    Increase value of $len_{C_{obj}}$ by 1
    if $len_{C_{obj}}$ == window-length then
        Add the sequence $C_{obj}$ to the list $L_{obj}$
        Decrease value of $len_{C_{obj}}$ by 1
        Delete the first element from $C_{obj}$
foreach $obj$ in $uniq\text{-}objs$ do
    if $len_{C_{obj}} > 0$ then Add the sequence $L_{obj}$ to the list $L_{obj}$
Initialize $birthmark$ to empty set { }
foreach $obj$ in $uniq\text{-}objs$ do
    $birthmark$ = union of ($birthmark$, $L_{obj}$)
```

Algorithm 1: Computing Birthmark

```
Input: Birthmarks $B$ and $B_{obfs}$ of two apps, $A$ and $A_{obfs}$ respectively
Output: A similarity coefficient between the two apps, $A$ and $A_{obfs}$
Calculate $I$ = intersection of ($B$, $B_{obfs}$)
Calculate $U$ = union of ($B$, $B_{obfs}$)
Similarity score = $\frac{Numberofitemsin I}{Numberofitemsin U}$
```

Algorithm 2: Calculating Similarity

4 Implementation

Our implementation of the birthmark-based similarity detection approach consists of two parts. The first part is the system that profiles applications and collects execution traces and is described below. The second part is the implementation of the birthmark algorithm described in the previous section, and we do not describe it in further detail here.

The Android SDK ships with a default method profiling tool, which can be used to collect execution traces of an application running either on an Android device or on the Android emulator. We used this tool to collect run-time traces of apps executing in the Android emulator. We worked with Android SDK 2.3.7 (Gingerbread). In this section, we will describe the changes we made to the Android framework for trace collection and filtering.

– *Using default Android profiler* Dalvik Debug Monitor Server (DDMS) is a debugging tool provided in Android's SDK that also offers the ability to trace apps. Users can leverage DDMS to trace the execution of a running app as it performs arious activities using a simple user interface. The resulting file is a concatenation of data in binary format and textual information about mappings between the binary identifiers in the data and the corresponding method names. We wrote a C program to parse the trace file and output the sequences of methods invoked in the trace.

- *Modifications to the profiling code* We modified the source code of the default profiler to make two changes to the trace generation:
  1. *Tracing app-specific method calls* The default profiler in Android profiles all the method calls in the call hierarchy including those invoked by the application as well as the methods executed by the underlying Android framework and the Dalvik virtual machine. We are interested in logging only the method calls invoked by the application. So we modified the source files of the Android framework, so that the profiler logs a method call, only if the return address of the calling function falls within the memory boundaries at which the application is loaded. This ensures that the method call being logged was invoked from within the application and does not include method calls invoked outside from the application such as those made by the Android API.
  2. *Emitting object ID* For calculating API birthmarks, we need the ID of the object on which each method call is invoked during the application run, so as to segregate API method calls based on the respective invoked objects. Hence we modified the Android source so as to emit the object ID in each record of the generated trace. We log the following fields in the final trace: *thread ID, method name, method arguments and return type, class name, object ID*.
- *Trace filtering*

  There are three types of APIs that an Android application interacts with:
  1. Android framework API: Set of methods provided by the underlying Android framework
  2. Java standard API: Set of methods provided by the standard Java language implementation
  3. APIs that are part of the application package (including various advertisement libraries)

  During birthmark computation, we want to include only those method calls that are indispensable to the Android application. Hence we decided to log the interaction of the application with the underlying Android framework. We decided not to keep track of the interaction of the application with the libraries that come as part of the application package (such as advertisement libraries). Such APIs being internal to the application, are easy to replace with other equivalent APIs. For example, consider a free mobile application that uses multiple advertisement libraries to display advertisements to the users. An attacker can easily remove one of these APIs and repackage the application, without changing the application's functionality. Also, one can easily change the methods' names declared inside one of these APIs. The discussion about exclusion of Java API methods is given later in Sect. 5.8.

5 Evaluation

5.1 Goals

To evaluate the API birthmark algorithm, we need to answer the following question: What are the essential characteristics of a birthmark of a mobile app?

A birthmark should be able to detect high similarity between *identically behaving* copies of the same program, even if the source code of the two programs differs significantly (*e.g.,* because of applying various program transformations). As a corollary, if a large portion of the source code of two programs is the same, the birthmark should detect them as copies. Additionally, when given two dissimilar programs as input, it should be able to distinguish between them by giving low value of similarity. We conducted different experiments to evaluate the API birthmark algorithm on these three factors. We first present our evaluation setup which is followed by a description of these experiments.

5.2 Evaluation setup

The very first step in our evaluation was creation of the required dataset of app pairs. To assess the resilience of the birthmark against program transformations, we need a dataset consisting of two types of apps, a corpus of apps that implement a variety of functionalities, and the same apps, obfuscated using semantics-preserving transformations on the original apps. As explained in Sect. 2, we designed our own obfuscator, named Androcrypt, that would encrypt the app's binary, store the decryption logic as the first statement to be executed in the new app and thus produce an identically-behaving obfuscated app. We collected a corpus of Android apps, and obfuscated them using Androcrypt. We then ran the API birthmark algorithm on execution traces of all possible $(A, A'_{obfuscated})$ pairs of apps from this dataset. Each pair consists of an original app A and an app $A'_{obfuscated}$ obtained by obfuscating an original app A'. Here, two cases are possible: either $A' = A$ i.e., A' is the same app as A or $A' \neq A$ i.e., A' is a different app than A. In the first case when we run the algorithm on traces of $(A, A_{obfuscated})$ types of pairs, consisting of an original app A and its obfuscated counterpart $A_{obfuscated}$, we are testing the resilience of the API birthmark against obfuscations. In the second case when pairs are of type $(A, A'_{obfuscated})$ where $A'_{obfuscated}$ is obfuscated counterpart of an app A' which is *different* than A, we are testing the ability of API birthmark to distinguish between distinct apps.

To compute the API birthmark of an original app A or its obfuscated counterpart $A_{obfuscated}$, we need the corresponding runtime trace. Executing each app manually is time consuming and would impede the detection of plagiarized apps in a large collection (e.g., at app market scale), thereby limiting the scalability of the approach. Therefore, we decided to automate the process of Android app execution. For running Android apps without any manual intervention, we used a tool called Monkey that drives the app automatically by generating input events such as clicks and touches, as described in Sect. 5.10. We then collected the execution trace using DDMS as explained in Sect. 4. From our dataset of $(A, A'_{obfuscated})$ pairs of original apps and their obfuscated counterparts, the automatic execution succeeded for a total of 350 app pairs (downloaded randomly from Google's official Android app market).

We divided our dataset into two, a smaller set of 50 apps was used as the *training set* and the remaining apps formed an *evaluation set*. The *training set* was used to set the threshold for the similarity measure. Using this threshold, we evaluated the birthmark algorithm on the apps in the *evaluation set*. We also did a number of experiments on this dataset such as verifying the credibility of the API birthmark.

All our experiments were done on Linux machine with Intel i5 quad-core 3.10GHz processor, 8 GB RAM and running Ubuntu 12.04. On an average, it took 0.2 seconds to filter two traces and run API birthmark algorithm on one pair of apps.

5.3 Choosing a threshold value

Prior to performing the experimental evaluation, we first need to set the threshold for the similarity coefficient that is used to determine whether an app is plagiarized. As described in Sect. 5.2, we used the smaller training set of 50 apps to set the threshold. The 50 original apps and 50 obfuscated counterparts of these apps formed 50×50 $(A, A_{obfuscated})$ app pairs. On every pair in this set, we ran the API birthmark algorithm. We experimented with different values of threshold and calculated the number of false positives and false negatives, as reported in Table 6. The total number of wrong classifications is equal to the sum of false positives and false negatives, as shown in the last column of Table 6. One could choose to minimize the number of false positives alone or the number of false negatives alone. We chose to minimize the total number of wrong classifications, and hence decided to set the threshold value to 0.5, which gives the least number of wrong classifications as shown in the last column of Table 6.

We used this threshold to evaluate the API birthmark algorithm on the apps in the evaluation set.

5.4 Setting the window size

Another parameter to be set for the API birthmark algorithm is the window size that is nothing but the length of the API method sequences generated from the execution traces during the API birthmark calculation, as explained in Sect. 3.2.2. We used the training set to experiment with different windows sizes and calculated the wrong classifications done by the API birthmark algorithm. Table 7 shows the results.

These values show that choosing a smaller window size increases the number of false positives, which implies that the similarity between different programs increases where as a bigger window size leads to a rise in the number of false negatives, which suggests that the similarity between identical programs decreases. We chose to set the default window size to 3, so that the API birthmark algorithm can distinguish between different programs (less false positives), at the same time not having an unacceptable number of false negatives.

Table 6 Evaluation for different values of threshold

Threshold	False negatives	False positives	False negatives + false positives
0.2	2	87	89
0.3	5	46	51
0.4	13	9	22
0.5	16	2	18
0.6	18	2	20
0.7	21	0	21

Table 7 Evaluation for different window lengths

Window length	False negatives	False positives	False negatives + false positives
1	3	28	31
2	6	17	23
3	16	2	18
4	18	2	20

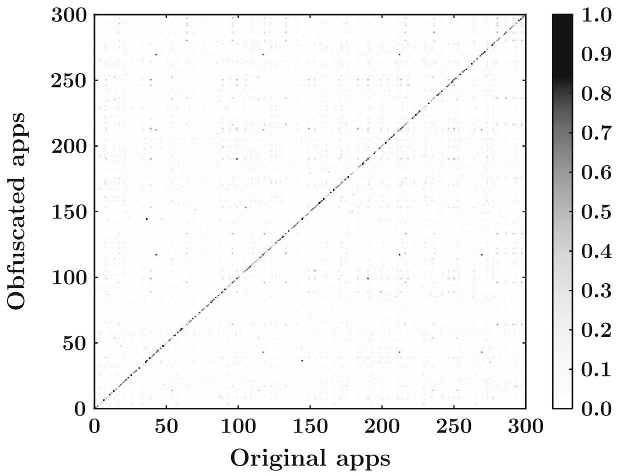

Fig. 2 API birthmark results on pairwise comparisons for 300 apps (300 × 300 pairs). Diagonal shows the result of comparing an app and its obfuscated counterpart. Non-diagonal shows comparisons between app and obfuscated counterpart of a different app

5.5 Detecting obfuscated apps

We have 300 original apps and 300 obfuscated apps produced by using the encrypting obfuscator described in Sect. 2. The total number of possible $(A, A_{obfuscated})$ pairs is therefore 300×300. We run the API birthmark algorithm for every pair, thus producing

Table 8 Evaluation of API birthmark algorithm for the results in Fig. 2

Total apps	False negatives	False positives
300	45 (15 %)	61 (20.3 %)

a matrix of dimensions 300×300. The results are shown in Fig. 2. A point on the diagonal point corresponds to similarity measure between an app and the obfuscated version of the same app, and therefore should have a value closer to 1. A non-diagonal point corresponds to similarity measure between an app and the obfuscated version of a different app, and should therefore have a value closer to 0.

We evaluated the similarity measures produced by the API birthmark by calculating the number of false positives and false negatives as shown in Table 8. The number of false negatives is nothing but the number of apps, A, for which the similarity coefficient produced by API birthmark for the pair $(A, A_{obfuscated})$ is less than the threshold and $A_{obfuscated}$ is the obfuscated counterpart of the same app as A (we chose a threshold value of 0.30; we discuss the computation of this threshold in Sect. 5.3). The number of false positives is the number of apps, A, for which similarity coefficient produced by API birthmark for the pair $(A, A_{obfuscated})$ is greater than the threshold and $A_{obfuscated}$ is the obfuscated counterpart of a different app than A.

There are three cases in which the dynamic API birthmark reports a large similarity coefficient between apps of a certain category in spite of the apps being distinct.

(1) *Customized apps* It is a common practice among app developers to release the same app multiple times, each one built under different package name, such that the core functionality of all packages is the same but the input configuration files are different for each package. An example would be different packages of the app, each one built to display the text in the app in a different language. Such apps have identical source code and differ only in the resource files such as text files or image files. As a result, the execution traces generated during execution of two such apps are nearly identical. API birthmark algorithm therefore detects high similarity between such apps. In our dataset, we found 16 apps in total that fall under this category. An example of one app is given below. This app is a puzzle game, released under three different packages, each one showing different quiz questions, but having identical structure. The classification of packages of

App name	Package name
How I Met Your Mother Trivia	com.pbgames.q.himym
Gossip Girl Trivia	com.pbgames.q.gg
Two and a Half Men Trivia	com.pbgames.q.tahm

two such apps as similar by the API birthmark is indeed truthful. Therefore, we removed such cases from the count of false positives. The table below gives the number of false positives after this filtering.

Total apps	False negatives	False positives
300	45 (15 %)	45 (15 %)

(2) *Use of programming framework* There are many programming frameworks such as PhoneGap that let developers write apps using web technologies. These frameworks interact with the underlying mobile operating system such as Android. Hence apps developed using such frameworks exhibit a common set of API method sequences that are part of this interaction. On encountering this common set of method sequences in the traces of such apps, API birthmark algorithm computes high similarity between them. To prevent such apps from being detected as similar, we can collect the set of such commonly found API method sequences and discard them during API birthmark calculation.

(3) *Use of common libraries* Many Android apps use a common set of libraries, such as Google's advertisement library. Because of the interaction of the advertisement library with Android, such apps also display a common API methods during their execution. A similar approach as mentioned above can be used to filter out the common method sequences and thus prevent the classification of the apps as similar.

We believe that the above two filtering techniques would further reduce the number of false positives observed.

5.6 Detecting identical apps

The goal of this experiment is to test the API birthmark's ability to detect copies of the same app. Since a birthmark of an app is its unique fingerprint, birthmarks of two identical copies of the app (such copies have the same source code too) should be the same. We executed each app twice, treating the resulting two traces as if generated by identical copies of the same app. For each of the two executions of the app, we gave the same seed value to the event generator of Monkey, thereby making sure that same input event sequences are generated for both runs of the app. This is to simulate the execution of two copies of the same app with the same user input.

We then ran the birthmark algorithm on the two traces of the same app, thereby computing two birthmarks, and then calculated the similarity coefficient between them. We repeated this procedure for every app in our dataset of 300 apps. The resultant values of similarity coefficient are on the diagonal of Fig. 3. The number of false negatives for results in Fig. 3 is the number of apps which the API birthmark algorithm failed to detect as similar, even though both the traces were generated from the same app. We observed that the number of false negatives dropped to 12 as compared to the experiment in Sect. 5.5 in which it was 45. This is an expected result, since we are comparing two traces of the same app here as opposed to comparing an app and an obfuscated app shown in Fig. 2.

5.7 Detecting distinct apps

As much as a birthmark should detect copies of apps, it is important that it should be able to distinguish between two different programs by indicating a low value of

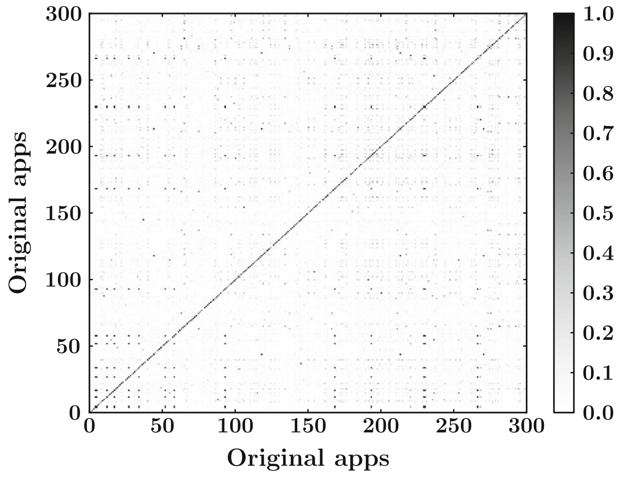

Fig. 3 API birthmark results on pairwise comparisons for 300 apps (300 × 300 pairs). Diagonal shows the result of comparing two traces of the same app. Non-diagonal shows comparisons between traces of two different apps

similarity between them, thereby proving its credibility. We tested this aspect with our dataset of 300 apps. We did a pairwise comparison for each pair (*A*, *B*) where *A* and *B* are two apps from our dataset and *A* is different from *B*. The results are as shown on non-diagonal points of Fig. 3.

The number of false positives is the number of apps for which the API birthmark algorithm produced similarity coefficient above the threshold during comparison of the app with at least one other app. We observed that the number of false positives increased for this experiment as compared to the experiment described in Sect. 5.5. It is equal to 115, and this number reduced to 86 after filtering out the different customized versions of the same app. The reasons behind this misclassification are same as those explained earlier in Sect. 5.5 such as presence of apps developed using a common programming framework and apps using the same advertisement library. Additional filtering techniques are needed to accommodate such apps.

5.8 Effect of inclusion of Java API methods

During the calculation of API birthmark, we are only observing the Android API method invocations by the app. But an Android app makes use of Java API methods as well. We wanted to evaluate the effect of including this usage of the Java API by the app. Figure 4 shows the result of pairwise comparison of 300 apps and 300 obfuscated counterparts of the apps, in which the collected traces have the Java API method calls in addition to the Android API method calls.

Collecting the Java API methods makes the individual traces larger, which in turn adds more items in the API birthmark calculation. As a result, the union of sets of method sequences computed for calculating the Jaccard similarity becomes larger, and hence it results in lower absolute value of similarity coefficient between two apps. For example, the median value of similarity coefficient for an app and obfuscated

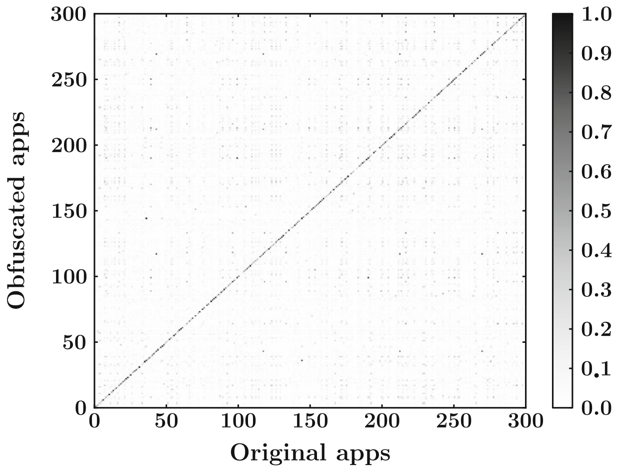

Fig. 4 API birthmark results on pairwise comparisons for 300 apps (300 × 300 pairs), keeping the Java API method call in the execution traces. Diagonal shows the result of comparing an app and its obfuscated counterpart. Non-diagonal shows comparisons between app and obfuscated counterpart of a different app

counterpart of a distinct app is 0.03 without Java and it is 0.04 with Java (the lower this value, the better it is). The median value of similarity coefficient for an app and obfuscated counterpart of the same app is 0.73 without Java, and the same value is 0.67 with Java.

We therefore concluded that inclusion of Java API methods doesn't have an absolute effect on the birthmark computation. We decided not to include the Java API methods in our birthmark computation, since it results in lower values similarity coefficient.

5.9 Experiments with ProGuard as obfuscator

In our experiments so far, we used Androcrypt as the obfuscator to produce the obfuscated counterparts of original apps. As shown in Sect. 2, Androcrypt is the strongest obfuscator that we know of and hence produces strong obfuscated code. The stronger the obfuscation, the lower the similarity measures reported by any similarity detector. As a corollary, the weaker the obfuscation, the higher the similarity measures. Hence, if we use any other obfuscator which is weaker than Androcrypt, the scores reported by our API birthmark technique should only get higher. To confirm this hypothesis, we performed the following experiment. We took the same set of 53 apps shown in the Appendix, and used off-the-shelf obfuscator—ProGuard—to obfuscate the apps. We ran API birthmark technique on the pair of apps $(A, A_{obfuscated})$ where A is an original app and $A_{obfuscated}$ is same app obfuscated using ProGuard. We then ran the API birthmark on each pair of apps. The results are shown in the column labelled as (PG, B) in Appendix. For comparison, we also give values reported by API birthmark when Androcrypt was used instead of ProGuard in the column labelled as (AC, B) in Appendix. We show box plot of the values in Fig. 5. As the plot shows, when an off-the-shelf obfuscator such as ProGuard is used, API birthmark performs at least as good as or better, compared to the case when Androcrypt is used.

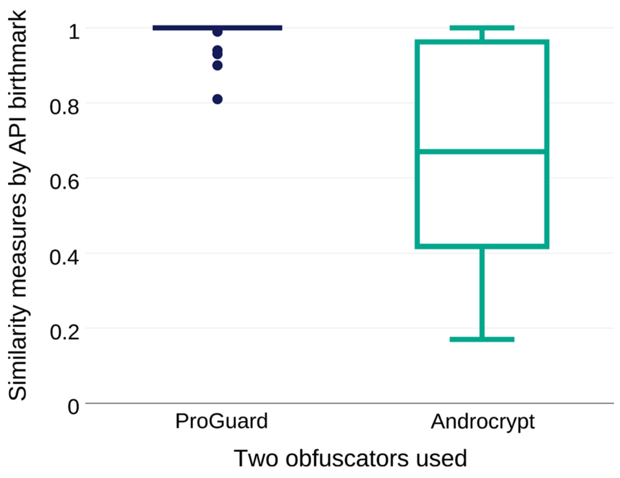

Fig. 5 Distribution of similarity measures reported by API Birthmark when two different obfuscators were used. *Box on left* corresponds to using ProGuard as the obfuscator whereas *box on right* corresponds to use of Androcrypt as obfuscator. The values used to draw the plots are taken from the columns labelled as (PG, B) and (AC, B) in Appendix

5.10 Automated execution using Monkey

The API birthmark is a dynamic birthmark that requires run-time trace generated during execution of the app. Since Android apps are interactive in nature, you need to provide user inputs such as clicks, gestures and touch events to them on a continuous basis during their run. During manual execution of the app, the user of the app provides these inputs. But manual execution is time consuming and limits the number of apps that one can experiment with. Therefore, we wanted to automate the execution of Android apps.

In order to run the apps automatically, you need a tool that will generate events automatically as opposed to manually supplying the events. We found that the Monkey program shipped with Android SDK is one of the best tools available for such purposes. The primary purpose of Monkey is to stress-test the app by simulating the generation of various input events such as clicks and touches. But by using this automatic generation of input events, you can automate the execution of an Android app. Monkey can be configured to run with a number of command line options. For our setup, we used the following options: the seed value of random number generator, total number of input events to be generated and option to ignore the exceptions during app's execution. For a given app and a seed value, monkey generates a particular but random sequence of events. We give the same seed value for execution of both, the original app and the plagiarized app, so that the exact same operations are exercised while executing the pair of apps using monkey.

Using Monkey, we succeeded in automatic execution and trace collection for 350 apps. Use of Monkey for more apps in our dataset failed due to a number of reasons. The events supplied by Monkey may not be context-sensitive i.e., they may not be relevant for the current execution state of the app, resulting into the app's crash, thereby producing no trace. Monkey does not give any preference to frequently occurring

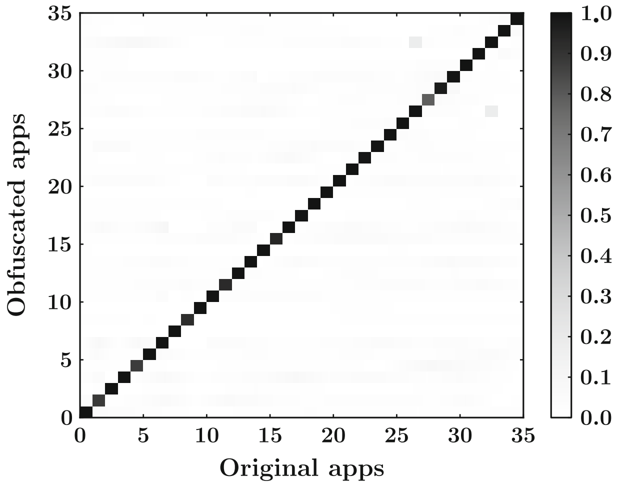

Fig. 6 API birthmark results on pairwise comparisons for 35 apps (35 × 35 pairs) with traces generated manually. Diagonal shows the result of comparing an app and its obfuscated counterpart. Non-diagonal shows comparisons between app and obfuscated counterpart of a different app

events over infrequent ones during the event generation. Moreover, it is hard to predict the right fraction of UI events and system events needed to run the app exhaustively, that can be given as input to Monkey. This results in poor coverage of the app's functionality, thereby leading to poor quality of the generated traces. The incomplete coverage of the app's functionality in traces directly affects the computation of the API birthmark. We conducted an experiment to study the impact of using Monkey for the collection of execution traces of Android apps. We chose 35 apps randomly from our dataset. In addition to the traces that were generated earlier using Monkey for these apps, we manually executed these apps by interacting with the app as a user and collected another set of traces. Figures 6 and 7 shows the results of pairwise comparisons of these apps and their obfuscated counterparts, with manually generated traces and traces generated using Monkey, respectively. It is evident from the two graphs that the API birthmark algorithm gives more accurate similarity coefficients when the traces have been generated manually. We therefore believe that use of a more robust tool for automated execution of Android apps would give better results. In future, we plan to repeat our experiments using other tools available such as Dynodroid (Machiry et al. 2013) which has better coverage of the app's functionality than Monkey. However, there are trade-offs of choosing each of these tools *e.g.,* Dynodroid suffers from a performance penalty (Dynodroid is 5X slower than Monkey) and hence may not be suitable for app execution at a large scale.

5.11 Attacks and limitations

Let us now look at the possible attacks on the API birthmark. An attacker can inject random API method calls in the source code of the app and thus skew the API birthmark. Such API method call injection may be done by employing techniques used in the creation of polymorphic viruses (You and Yim 2010). For this attack to be

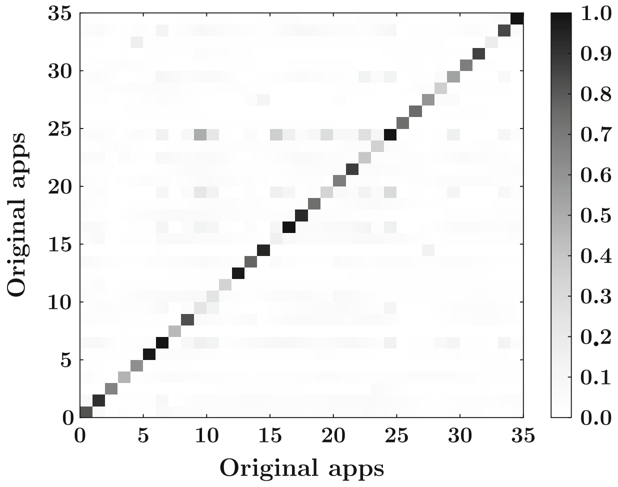

Fig. 7 API birthmark results on pairwise comparisons for 35 apps (35 × 35 pairs) with traces generated automatically using Monkey, keeping the Java API method call in the execution traces. Diagonal shows the result of comparing an app and its obfuscated counterpart. Non-diagonal shows comparisons between app and obfuscated counterpart of a different app

effective, each newly added method call or the method calls added as a group should produce zero side effects. Inferring the dummy methods from the API *automatically* is a hard problem. The attacker will have to resort to manual techniques for finding such API methods or method sequences. Indeed, if discovery of such methods or method sequences is an easy task, we can generate them ourselves, and filter them out from the dynamic traces, before running the API birthmark algorithm. Moreover, the attacker will have to insert at least 30 percent new sequences (since similarity threshold is 0.3), which leads to increased code size and additional runtime cost. In general, the attacks on the API birthmark have cost overhead and involve manual work which offer less incentives for the attacker to implement them.

First limitation of our approach is its scalability. In Sect. 5.10, we explain the difficulties faced in automating the execution of Android apps. Even the best tool among all of the available ones, namely *monkey*, does not yield useful traces for a number of available apps because of many reasons such as crashes in the middle of the execution or generation of extremely short traces. Also, the encrypting obfuscator works only partially on apps that contain certain classes such as *ContentProvider*, as described in Sect. 2. These factors put a limit on the number of apps with which we can experiment. Unless the automatic execution of Android apps becomes pratical for large datasets, experimenting on them remains a subject of future work. One practical solution could be to run the static analyses first on the larger dataset of apps which would narrow down the suspected plagiarized apps. We can then run the API birthmark algorithm on the smaller dataset.

Our setup includes automatic execution of apps by using the monkey program shipped with Android SDK. The fact that we rely on monkey for generating input events has two implications.

1. Monkey generates a random sequence of input events corresponding to a seed value. Greater the coverage of input events provided by monkey, better it is to compute similarity using API birthmark algorithm. To increase the runtime test coverage, better tools are needed for automatic testing of Android apps.
2. For a given seed value, monkey generates a stream of events such as clicks, touches or gestures to stress test the app. We give the same seed value to both, the original and the plagiarized app, so that the same sequence of events is generated while executing them. However, if there are small UI tweaks in the plagiarized app, some of the generated events would be impossible to operate on the plagiarized app. For example, if monkey has generated an event of type *Send touch-event-at-(x,y)=(1.0, 4.0)*, but the plagiarized app has moved the button to a different location, then an attempt to execute this operation may lead to unexpected consequences. Event generation in such cases may be accomplished by coupling monkey with other powerful GUI automation approaches such as Sikuli Script (Yeh et al. 2009) which provides a mechanism to programmatically control GUI elements in the automation scripts using their screenshots. Using Sikuli Scripts, one can use screenshots of GUI elements in the testing tool rather using raw coordinates, thus eliminating the scenarios where there are minor modifications in the GUI.

6 Related work

6.1 Birthmark-based software theft detection

The technique of constructing *software birthmarks* was proposed earlier by other researchers in the context of traditional desktop programs. To our knowledge, we are the first to use software birthmarks on mobile applications. Software birthmarks can be categorized into two classes, dynamic and static. Myles and Collberg introduced the whole-program-path dynamic birthmark (Myles and Collberg 2004). Our approach of using dynamic API birthmarks has been built upon the techniques proposed in Tamada et al. (2004) and Schuler et al. (2007). Tamada et al. (2004) proposed a dynamic API birthmark based on observations of the interaction of windows application with its environment. Schuler et al. (2007) put forward an improved version of dynamic API birthmark that is based on watching the program interaction at the level of objects which results in shorter API sequences. These two projects were done in the context of traditional desktop software (Windows applications and Java programs, respectively). We have applied those ideas to the domain of mobile apps, particularly for the problem of identifying plagiarized mobile apps.

A slightly different problem of assessment of similarity between two algorithms was targeted in Zhang et al. (2012) where they used two dynamic value-based approaches, namely N-version programming and annotation.

Among different bodies of work that use static birthmarks, GP_{LAG} (Liu et al. 2006) is a tool to detect plagiarism in software by mining program dependence graphs. A birthmark for Java applications was developed by Lim et al. (2008) that identifies and uses possible stack patterns that may be formed during program execution by analyzing the Java bytecode statically. Use of opcode-level k-grams as software birthmarks was done by Myles and Collberg (2005).

6.2 Code clone detection

The problem of detecting clones in software code has been well studied. A survey paper by Bellon et al. (2007) gives a good comparison and evaluation of various clone detection tools for traditional software programs. Various techniques have been explored to detect code clones that derive and use different type of information from the code such as text and tokens (Higo et al. 2002), metric vectors (Kontogiannis et al. 1995), abstract syntax trees (Baxter et al. 1998) and program dependency graphs (Krinke 2001).

6.3 Detecting plagiarized mobile apps

The problem of identifying plagiarism in mobile apps has attracted a lot of attention recently. Zhou et al. (2012), the idea was to generate a unique hash of the app from its Dalvik bytecode by using a fuzzy hashing technique. Zhou et al. (2013), the authors use a number of features of the application such as the android API methods, permissions requested by the application etc. to construct a feature vector and then employ Jaccard distance to define distance between two feature vectors. Crussell et al. (2012), the approach is to first group together similar apps based on certain features and then apply program-dependence-graph-based techniques to detect cloning.

To summarize, the proposed approaches employ different techniques such as fuzzy hashing (Zhou et al. 2012), feature hashing (Hanna et al. 2012), program dependence graph (PDG) (Crussell et al. 2012, 2013) and module decoupling (Zhou et al. 2013) for computing unique fingerprint of the app. These techniques rely on extracting static properties of the application by analyzing the application's source code. These can be defeated easily by code obfuscations. We are proposing an effective plagiarism detection technique for mobile apps based on dynamic analysis which is resilient to code obfuscations.

7 Summary

Code plagiarism is an important problem that plagues the mobile app development community, and serves as a popular vehcile for the delivery of malicious apps. Although the community has developed a number of code similarity metrics to combat plagiarism, they rely on syntactic features of the code to operate.

We show that such syntactic similarity measures are broken, because they can easily be evaded using simple obfuscations. We have developed a robust API birthmark-based approach to detect code similarity. Experiments on a dataset of Android apps shows that the birthmark-based approach is effective at detecting code plagiarism even with obfuscated apps.

Acknowledgments This work is supported in part by NSF Grants 0952128, 1117711, 1420815, 1441724 and 1408803.

Appendix

See Table 9.

Table 9 This appendix presents the similarity scores reported by two app similarity detectors, Androguard and API Birthmark, when app and its obfuscated version are given as input

Category	App/package name	App size (in KB)	(PG,AG)	(AC,AG)	(PG,B)	(AC,B)
Development	`alogcat`	40	0.76	0.03	1.00	0.68
Games	`chessclock`	92	0.78	0.09	1.00	0.74
Games	`tictactoe`	1963	0.24	0.01	0.93	0.57
Games	`solitaire`	70	0.84	0.01	1.00	1.00
Games	`sokoban`	109	0.82	0.07	1.00	0.88
Games	`kaesekaestchen`	148	0.79	0.05	1.00	0.69
Games	`atomix`	210	0.90	0.02	1.00	0.23
Games	`lexic`	245	0.68	0.02	0.81	0.26
Games	`androc`	539	0.79	0.01	1.00	0.17
Games	`games.memory`	2080	0.68	0.03	0.90	0.25
Games	`bomber`	455	0.80	0.11	1.00	0.67
Games	`blockinger`	801	0.48	0.01	1.00	0.17
Games	`blokish`	421	0.71	0.02	1.00	1.00
Games	`amazed`	15	0.44	0.10	1.00	0.67
Games	`opensudoku`	211	0.95	0.01	1.00	0.75
Internet	`blitzmail`	280	0.22	0.00	1.00	1.00
Internet	`connectbot`	858	0.72	0.00	1.00	0.70
Internet	`reddit`	759	0.47	0.00	1.00	0.50
Multimedia	`binauralbeat`	965	0.63	0.02	1.00	0.41
Multimedia	`avs234`	162	0.79	0.02	1.00	1.00
Multimedia	`zooborns`	39	0.75	0.05	1.00	0.41
Multimedia	`zxing`	720	0.61	0.00	1.00	0.95
Multimedia	`photostream`	134	0.77	0.03	0.99	0.61
Navigation	`pedometer`	46	0.97	0.03	1.00	0.39
Navigation	`stardroid`	2188	0.53	0.51	1.00	0.96
Office	`aarddict`	1852	0.29	0.00	1.00	0.65
Office	`babycaretimer`	457	0.23	0.01	1.00	0.46
Office	`calculator`	77	0.66	0.01	0.99	0.56
Office	`coinflip`	422	0.67	0.04	0.99	0.97
Office	`birthdroid`	79	0.78	0.08	1.00	0.85
Office	`Keyer`	82	0.65	0.04	1.00	0.40
Office	`TeaTimer`	216	0.77	0.04	1.00	0.51
Office	`aGrep`	54	0.88	0.04	1.00	1.00
Office	`simplydo`	64	0.73	0.03	1.00	0.56
Office	`tipitaka`	502	0.49	0.00	0.99	0.77
Office	`mileage`	366	0.72	0.56	1.00	0.96
Office	`wikinotes`	119	0.92	0.28	1.00	0.31

Table 9 continued

Category	App/Package name	App size (in KB)	(PG,AG)	(AC,AG)	(PG,B)	(AC,B)
SMS	`autoanswer`	88	0.54	0.14	1.00	1.00
SMS	`autoawayy`	112	0.69	0.03	1.00	1.00
Reading	`adsdroid`	116	0.65	0.01	1.00	1.00
Reading	`andquote`	38	0.88	0.06	1.00	0.24
Education	`antikythera`	531	0.71	0.08	1.00	0.83
Education	`angulo`	18	0.93	0.08	1.00	1.00
System	`airpushdetector`	33	0.86	0.13	1.00	0.65
System	`autostarts`	288	0.43	0.00	0.90	0.42
System	`appalarm.pro`	121	0.80	0.01	1.00	0.22
System	`apptracker`	140	0.57	0.02	0.94	0.46
System	`asqlitemanager`	339	0.71	0.01	1.00	0.23
System	`batterydog`	21	0.73	0.38	1.00	1.00
System	`httpmon`	74	0.80	0.01	1.00	1.00
System	`adbWireless`	378	0.36	0.00	1.00	0.92
System	`androsens`	21	0.91	0.21	1.00	0.54

The first column lists the category of the app. The second column gives the app package name. The last four columns show results for different combinations of obfuscators and similarity detectors. The abbreviations in the four columns stand for the following: (PG, AG): ProGuard as obfuscator, Androguard as similarity detector. (AC, AG): Androcrypt as obfuscator, Androguard as similarity detector. (PG, B): ProGuard as obfuscator, API Birthmark as similarity detector. (AC, B): Androcrypt as obfuscator, API Birthmark as similarity detector

References

Allatori Java obfuscator. http://www.allatori.com/

Androguard. http://code.google.com/p/androguard/wiki/Usage#Androsim

Baker, B.S.: On finding duplication and near-duplication in large software systems. In: WCRE (1995)

Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: ICSM (1998)

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Trans. Softw. Eng. **33**(9), 577–591 (2007)

Christodorescu, M., Jha, S.: Testing malware detectors. In: ISSTA (2004)

Crussell, J., Gibler, C., Chen, H.: AnDarwin: Scalable detection of semantically similar android applications. In: ESORICS (2013)

Crussell, J., Gibler, C., Chen, H.: Attack of the clones: detecting cloned applications on android markets. In: ESORICS (2012)

Dalvik Debug Monitor Server (DDMS). http://developer.android.com/tools/debugging/ddms.html

Dalvik Debug Monitor Server (DDMS). http://docs.eoeandroid.com/tools/debugging/debugging-tracing.html

DashO Java obfuscator. http://www.preemptive.com/products/dasho/overview

Ducasse, S., Nierstrasz, O., Rieger, M.: On the effectiveness of clone detection by string matching. J. Softw. Maint. **18**(1), 37–58 (2006)

Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A survey of mobile malware in the wild. In: SPSM (2011)

FOSS apps for Android. https://f-droid.org

Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., Choi, H.: Adrob: examining the landscape and impact of android application plagiarism. In: MobiSys (2013)

Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., Song, D.: Juxtapp: a scalable system for detecting code reuse among android applications. In: DIMVA (2012)
Higo, Y., Ueda, Y., Kamiya, T., Kusumoto, S., Inoue, K.: On software maintenance process improvement based on code clone analysis. In: PROFES (2002)
Jaccard index. http://en.wikipedia.org/wiki/Jaccard_index
Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. **28**(7), 654–670 (2002)
Kontogiannis, K., de Mori, R., Bernstein, M., Galler, M., Merlo, E.: Pattern matching for design concept localization. In: WCRE (1995)
Krinke, J.: Identifying similar code with program dependence graphs. In: WCRE (2001)
Lim, H.I., Park, H., Choi, S., Han, T.: Detecting theft of java applications via a static birthmark based on weighted stack patterns. In: IEICE (2008)
Liu, C., Chen, C., Han, J., Yu, P.S.: GPLAG: detection of software plagiarism by program dependence graph analysis. In: KDD (2006)
Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: an input generation system for android apps. In: ESEC/FSE 2013 (2013)
Myles, G., Collberg, C.S.: Detecting software theft via whole program path birthmarks. In: ISC (2004)
Myles, G., Collberg, C.: K-gram based software birthmarks. In: SAC (2005)
PhoneGap. http://phonegap.com/
ProGuard. http://proguard.sourceforge.net/
Rastogi, V., Chen, Y., Jiang, X.: DroidChameleon: evaluating Android anti-malware against transformation attacks. In: ASIACCS (2013)
Schuler, D., Dallmeier, V., Lindig, C.: A dynamic birthmark for java. In: ASE (2007)
Tamada, H., Okamoto, K., Nakamura, M., Monden, A., Matsumoto, K.I.: Dynamic software birthmarks to detect the theft of windows applications. In: ISFST (2004)
UI/Application exerciser monkey. http://developer.android.com/tools/help/monkey.html
Yeh, T., Chang, T.-H., Miller, R.C.: Sikuli: using gui screenshots for search and automation. In: UIST (2009)
You, I., Yim, K.: Malware obfuscation techniques: a brief survey. In: BWCCA (2010)
Zhang, F., Jhi, Y.-C., Wu, D., Liu, P., Zhu, S.: A first step towards algorithm plagiarism detection. In: ISSTA (2012)
Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: IEEE Symposium on Security and Privacy (2012)
Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: CODASPY (2012)
Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of "piggybacked" mobile applications. In: CODASPY (2013)