Effects of Code Obfuscation on Android App Similarity Analysis

Jonghwa Park[1], Hyojung Kim[1], Younsik Jeong[1], Seong-je Cho[1], Sangchul Han[2], and Minkyu Park[2*]
[1]*Dankook University, Yongin, Korea*
{72150262, 72150251, jeongyousik, sjcho}@dankook.ac.kr
[2]*Konkuk University, Chungju, Korea*
{schan, minkyup}@kku.ac.kr

Abstract

Code obfuscation is a technique to transform a program into an equivalent one that is harder to be reverse engineered and understood. On Android, well-known obfuscation techniques are shrinking, optimization, renaming, string encryption, control flow transformation, etc. On the other hand, adversaries may also maliciously use obfuscation techniques to hide pirated or stolen software. If pirated software were obfuscated, it would be difficult to detect software theft. To detect illegal software transformed by code obfuscation, one possible approach is to measure software similarity between original and obfuscated programs and determine whether the obfuscated version is an illegal copy of the original version. In this paper, we analyze empirically the effects of code obfuscation on Android app similarity analysis. The empirical measurements were done on five different Android apps with DashO obfuscator. Experimental results show that similarity measures at bytecode level are more effective than those at source code level to analyze software similarity.

Keywords: code obfuscation, Android app, software similarity, software birthmark, reverse engineering

1 Introduction

Recently, more and more mobile devices such as smart phones and tablets are widely used and the mobile application (or app) market is growing rapidly. There are more than 1,800,000 Android apps in the Google Play on Dec 2015 [1]. As the mobile app market grows, software theft such as unauthorized duplication, distribution, and plagiarism is also increasing fast. According to Arxan's research, over 90% of top mobile apps have been hacked, cracked, and breached, and are available as illegitimate versions on third-party sites [2].

One of reasons why so many mobile apps have been stolen is that Android apps are based on Java programming language. Java applications, especially small-scaled applications such as mobile apps, can be easily reverse engineered and their source codes can be extracted using tools such as dex2jar, JD-GUI and APKtool. Attackers can steal essential techniques from the extracted source codes or insert malicious codes to produce malicious apps. There is a need to detect software theft on Android apps and prevent the distribution of repackaged Android apps.

To detect Android app theft, there are research approaches on the similarity analysis of Android apps based on software birthmark [3, 4]. Software birthmark is an intrinsic characteristics of an application and is extracted from its execution codes. Software birthmark can used to identify software or to detect software theft by analyzing similarity between birthmarks. Android apps are distributed as a single file in the format of APK (Android Application Package), which contains an executable file called *classes.dex*

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, volume: 6, number: 4, pp. 86-98
*Corresponding author: Department of Computer Engineering, Konkuk University, 268 Chungwondaero, Chungju-si, Chungcheongbuk-do, 27478, Korea, Tel: +82-43-840-3559

(hereafter referred to as .dex file). The software birthmark of an Android app can be extracted from its .dex file.

Code obfuscation is a technique that obfuscates a program's source codes or execution codes to conceal its purpose or logic, in order to prevent tampering or deter reverse engineering. Android app developers applies code obfuscation to protect their codes using tools such as Proguard [5], DashO [6, 7], and DexProtector [8]. On the other hand, code obfuscation can be also leveraged to bypass software theft detection that uses software birthmark based similarity analysis. In this case, the existing software birthmark based similarity analysis technique may fail to detect software theft.

In this paper, we analyze the effect of code obfuscation on the similarity of Android apps. We collect several open source Android apps from SourceForge.net [9] and obfuscate them using DashO obfuscator. First, we measure and analyze the similarity between the decompiled source codes of original APK and obfuscated APK using MOSS [10], which is a well-known source code-level software plagiarism detection system. Second, software birthmark is extracted from Java bytecodes of original APK and obfuscated APK, and the similarity between them is measured using Stigma [11], which is a well-known Java birthmarking tool. Experimental results show that similarity measures at bytecode code level are more effective than those at source level when codes are obfuscated.

The rest of this paper is organized as follows. We explain related work in Section 2. Section 3 briefly describes the method of our experiments. In section 4, the experiments results and analysis are presented in detail. We conclude this paper in Section 5.

2 Related Work

2.1 Java obfuscation techniques

Reverse engineering of software code by adversaries may reveal important technological secrets such as core algorithms and sensitive data. Code obfuscation is a technique that transforms the source or machine code of the program into a form difficult to be analyzed by reverse engineering [12, 13, 14]. Code obfuscation can be understand as a special case of data coding and is similar to code optimization. It tries to maximize obscurity while minimize execution time. According to several research groups [12, 13, 14, 15, 16], the types of code obfuscation have been classified into the following transformations: layout obfuscations, control obfuscations, data obfuscations, preventive obfuscations, and string encryption. Table 1 shows the types of code obfuscations.

As one example of lexical obfuscations, renaming identifier changes a name of a class, a variable, and a method into a meaningless one. This makes it difficult to extract useful information from an identifier when an attacker tries to analyze the reverse engineered code. Shrinking (Pruning) is to remove unused classes, methods or fields. If an application employs third party libraries, many classes or methods can be reduced or deleted.

Data obfuscation makes data scrambled to prevent unauthorized access to sensitive materials. Data obfuscation methods have been classified into three main groups: (1) Storage and encoding obfuscation, (2) Aggregation obfuscation, and (3) Ordering obfuscation. Examples of storage and encoding obfuscation are splitting variables, promoting scalars to objects, and changing a variable lifetime, Examples of aggregation obfuscation are merging scalar variables, modifying inheritance relations, and splitting or merging arrays. Examples of ordering obfuscation are reordering methods and reordering arrays. This form of encryption results in unintelligible or confusing data [17].

Control obfuscations make a control flow of a program difficult to understand. These transformations may rely on the existence of opaque predicates that are conditional statements whose predicates always

Table 1: Types of code obfuscation [12, 13, 15, 16]

Type	Description
Layout obfuscations (Lexical obfuscations)	As a one-way transformation, these transformations involve removing the source code formatting information and changing the lexical structure of the target program. These transformations include scrambling of identifiers (renaming identifiers) and changing formatting, removing of comments, and shrinking.
Data obfuscations	Transformations of data structures in order to obscure the purpose of program data fields. These transformations may be divided on three main groups: (1) Storage and encoding obfuscation – change of representation and methods of usage of variables, (2) Aggregation obfuscation – merging independent data and splitting dependent data, and (3) Ordering obfuscation – reordering of internal objects layout.
Control obfuscations	These transformations make it difficult to understand control flow in individual program functions. These obfuscations may be classified into three main groups: (1) Computation obfuscation – changing structure of control flow, (2) Aggregation obfuscation – merging and splitting fragments of code, and (3) Ordering obfuscation – Reordering of code blocks, loops and expressions with preserving their dependencies.
Preventive obfuscations	The main goal of these transformations is to make known automatic deobfuscation techniques more difficult, or to explore known problems in current deobfuscators or decompilers, not to obscure the program to adversaries.
String encryption	Replacing string literals with calls to a method that decrypts its parameter makes an adversary's life more interesting, but, not much. Because the strings must be decrypted at run time, the respective code must be contained in the application.

evaluate to true or false. Given opaque predicates, we can construct obfuscating transformations that break up the flow-of-control of a class method. The obfuscation techniques include API hiding, etc.

The common tools for obfuscating Android App are Proguard [5], DashO [6, 7], DexProtector [8], and the like. The available obfuscating techniques of three tools are summarized in Table 2 [18]. Proguard is a default obfuscation tool provided with the Android SDK package. Proguard removes unused classes, fields, methods, and attributes (shrinking); optimizes bytecode and removes unused instructions (optimization); and renames paths, classes, fields, methods, and variables using the alphabet by default (renaming). DexProtector is a commercial obfuscator for Android platform that protects Android applications against illegal use, reverse engineering, and cracking. It provides the obfuscation techniques such as string encryption, control flow obfuscation, class encryption, native code encryption, and API hiding with all functionality of Proguard.

DashO gives enterprise-grade protection for all Java and Android applications, reducing the risk of piracy, code theft, and tampering [18]. By allowing for statical analysis of the code to find the unused types, classes, methods, and fields, DashO provides the obfuscation techniques such as renaming of the

names of methods and variables, optimization with pruning, and encryption of strings in sensitive parts of the applications. In this paper, we use DashO to obfuscate Java program because it provides several transformation techniques. We can also apply the available techniques one by one (see Table 3).

Detection of code theft in Android app uses information from identifier, methods, fields, and Android API. These Android obfuscation tools change this information radically and accuracy of detection decreases. In other words, the obfuscation tools make similarity measure of two software more difficult and we need more research on effects of obfuscation.

2.2 Java similarity measures

Java applications are more vulnerable to reverse engineering than other language applications such as C, C++, and the like. Because most work is done in the Java standard library and the size of the user code is relatively small, attackers decompile the bytecodes and analyze the decompiled source codes in less time [17]. Android apps are written in Java and also vulnerable to reverse engineering attacks.

Software infringement occurs more often in Android environment due to the same reason. To prevent this infringement technically, many researchers are studying similarity measures. Software birthmark is a key characteristic that can detect algorithm theft or code theft. Two kinds of software birthmarks exist. Static birthmarks rely on information statically extracted from program such as initial values of the fields [19, 20]. Dynamic birthmarks, in contrast, rely on the information that can be gathered during the execution of the program [21].

Tamada et al. [19] proposed four characteristics of a Java class for static birthmarks: constant values in field variables (CVFV), sequence of method calls (SMC), inheritance structure (IS), and used classes (UC). CVFV birthmark is a (type, value) sequence of all field variables of a class. SMC birthmark is a sequence of methods appeared in a class and this order is not necessarily execution order. Methods belong to well-known classes, such as J2SDK and Jakarta project. IS birthmark represents the sequence of super classes of a user defined class. In this sequence, only well-known classes appear. UC is a set of classes used in a class and arranged in alphabetical order. UC also consider only well-known classes. The authors define Similarity of two classes a percentage of elements in two birthmarks.

Myles and Collberg proposed k-gram based static birthmark [20]. K-gram is a contiguous substring of length k of letters, words, or opcodes and the authors deal with k-gram of opcodes. K-gram is computed for each method and summed up for entire program. K-gram birthmark does not reflect the order and frequency of k-grams to be less susceptible to semantic-preserving transformations.

Myles and Collberg also proposed a dynamic birthmark called whole program path (WPP)[21]. WPP uses a dynamic control flow graphs (DCFG) of a program obtained during execution. It collects all the

Table 2: Java obfuscation tools

	Proguard	DashO	DexProtector
Shrinking	O	O	O
Optimization	O	O	O
Renaming	O	O	O
String encryption	X	O	O
Class encryption	X	X	O
Control flow	X	O	X
Method call hiding	X	X	O

Table 3: The obfuscation techniques of DashO

Option	Description
Removal (Pruning)	This removes unused classes in an application.
Renaming identifiers	Renaming classes, Renaming fields, Renaming methods (Overload induction), Renaming interfaces
Byte code optimization	This optimizes the code, and makes it smaller and faster.
String encryption	This makes it possible to encrypt string in sensitive parts of the application.
Control flow obfuscation	This adds confusion to the source code. This also adds confusion to the actual structure of the program.

compact DCFG and regards them as a program's birthmark. However, WPP birthmark may be unsuitable for large-scale programs due to overwhelming volume of WPP traces.

3 Methods of Experiments

We would like to know the effects of obfuscation on Android APK files and finally on measuring similarity between Android APKs. In order to analyze the effect of obfuscation, we decided to compare decompiled source codes of APKs. First, Android obfuscation are mainly applied to .class files during app packing. The obfuscation occurs while .dex files are produced from .class files. Input to the tools is APK file and output is obfuscated APK file, too. Second, the source codes comparison is very limited because in most cases we cannot obtain source codes of apps. Third, JAVA bytecodes contain enough information to steal codes from other apps and most attacks target to bytecodes.

We compared the decompiled source codes of the original and obfuscated APK. When we compared the two decompiled source codes, they differ from each other significantly at first. An APK file includes many files automatically generated during compilation and packaging. Most of these files are also decompiled together. An attacker, however, are not interested in these files and most bread-n-butter codes. We tried to exclude them when we compare them and analyzed the results.

4 Results and Analysis

When you build an Android app in Eclipse or Android Studio, the app is first compiled as Java application [22]. Input to `javac` is application source codes and ones automatically generated by Android platform. The automatically generated source codes include R.java (comes from resource files), BuildConfig.java, android.support.v4, and service interface files (AIDL). After applying `javac`, the `dx` tool of Android SDK converts the Java jar file into a classes.dex file.

We used AndroChef Java Decompiler for decompilation [23]. Table 4 summarizes the number of total lines of the five apps for experiments. The number in the third and fourth columns represents that of the decompiled source codes without all the automatically generated codes. The obfuscated apps are applied all the techniques provided by DashO.

The original and obfuscated apps are still considerably different from each other in terms of the number of their decompided source codes. This might be due to such as removing unused codes and optimization of code in Java compile process. In addition, some codes are never restored entirely from

Table 4: Number of lines of original source and decompiled source.

App name	original source	decompiled source from original APK	decompiled source from obfuscated APK
OpenCamera_v.1.26	17394	8142	7037
OpenCamera_v.1.27	17653	8426	7029
Focal_v.1.0	19540	12790	1891
Connectbot	51053	30930	20997
ShoppingList	2999	2028	17026

bytecodes. For example, GeneratePubkeyActivity.java (10340 bytes) and HelpActivity.java (2303 bytes) of original source of Connectbot app are truncated after decompilation.

In this experiment, we analyzed the effect of the obfuscation of DashO in two ways. First, we measure the similarity between the decompiled source codes of the original and obfuscated APKs. Second, we measure the similarity between the bytecodes of the original and obfuscated APKs.

4.1 Results of comparison of decompiled source codes

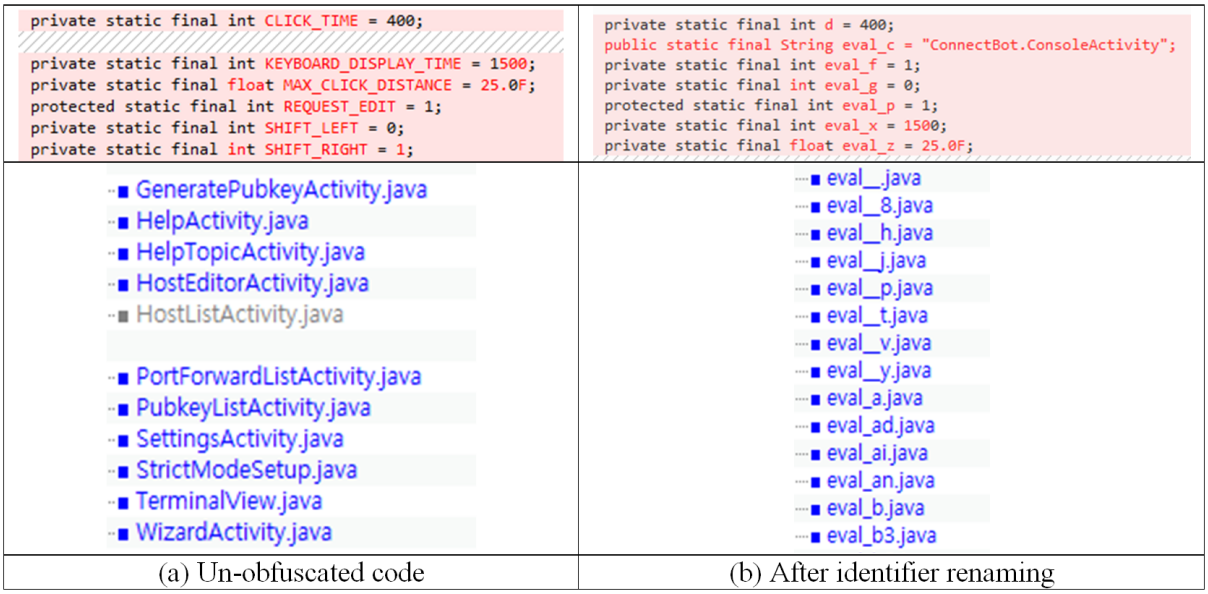
Table 5 shows the similarity of decompiled source codes of obfuscated APK with respect to decompiled source codes of original APK. We used MOSS [10, 24] as source code comparison tool.

Table 5: Similarity between decompiled source codes

Features	Opencamera 1.26	Opencamera 1.27	Focal	Connetbot	ShoppingList
Removal	68%	78%	81%	76%	83%
Renaming identifiers	66%	73%	19%	75%	76%
Bytecode optimization	86%	86%	91%	91%	92%
String encryption	71%	50%	67%	36%	68%
Control flow	51%	60%	50%	68%	37%
ALL	22%	26%	6%	38%	23%

The overall similarity is not so high because of obfuscation effects. Among the techniques, “Bytecode optimization” has little effect on the original APK because DashO performs algebraic identity, strength reduction, and other peephole optimizations. Figure 1 shows the result of applying “Renaming identifiers” obfuscation to Connectbot and Figure 2 does “String encryption”.

In the case of Focal app, “Renaming identifier” obfuscation decreases the similarity more significantly than other obfuscations. The number of lines of obfuscated Focal app is 3,364 and is much fewer than other obfuscated Focal app. For example, Bytecode optimized Focal app’s size is 14,077 line. In the case of Connectbot app, “String encryption” obfuscation decreases the similarity more significantly than other obfuscations. The number of lines of obfuscated Connectbot app is 24,355 and is much fewer than other obfuscated Connectbot app. Unlike Focal app, identifier renamed Connectbot’s size is 33,529 and is longer than other obfuscated APK.



(a) Un-obfuscated code (b) After identifier renaming

Figure 1: Comparison of decompiled sources from APKs with/without renaming

(a) Un-obfuscated code

```
public void onServiceConnected(ComponentName var1, IBinder var2) {
    ConsoleActivity.this.bound = ((TerminalManager.TerminalBinder)var2).getService();
    ConsoleActivity.this.bound.disconnectHandler = ConsoleActivity.this.disconnectHandler;
    Object[] var3 = new Object[]{Integer.valueOf(ConsoleActivity.this.bound.bridges.size())};
    Log.d("ConnectBot.ConsoleActivity",
     String.format("Connected to TerminalManager and found bridges.size=%d", var3));
    ConsoleActivity.this.bound.setResizeAllowed(true);
```

(b) After encrypting strings

```
public void onServiceConnected(ComponentName var1, IBinder var2) {
    int var3 = 0;
    ConsoleActivity.this.bound = ((TerminalManager.TerminalBinder)var2).getService();
    ConsoleActivity.this.bound.disconnectHandler = ConsoleActivity.this.disconnectHandler;
    Object[] var4 = new Object[]{Integer.valueOf(ConsoleActivity.this.bound.bridges.size())};
    Log.d(Exec.regionMatches(6, "EhfgohxOa{>R}}gzzrYznrjtjf"),
    String.format(R.indexOf("□><=16\"2<y.4|\t;--(,\"(\b\')).//9l, +p7=&:1v5*0><9.p,i{g>!a", 80), var4));
    ConsoleActivity.this.bound.setResizeAllowed(true);
```

Figure 2: Comparison of decompiled sources from APKs with/without string encryption

4.2 Results of comparison of bytecodes

We used Stigmata to compare bytecodes of APKs. Stigmata extracts birthmarks from each Java class files of two APKs and compares them pair-wisely [11]. Table 6 shows the implemented birthmarks in Stigmata used for the experiments. See the details of Stigmata in [11].

The steps of comparison are as follows:

1. Given Original.APK and Obfuscated.APK transformed by DashO
2. Generate Original.jar and Obfuscated.jar from the two APKs using dex2jar. The two jar files are input to Stigmata
3. Select one of the four birthmarks on Stigmata
4. Select comparison method from round robin, guessed comparison pair by its name, specified comparison pair, and filtering the result of round robin comparison method

Table 6: Some implemented Birthmarks in Stigmata

Birthmark types	Descriptions
SMC	Sequence of Method Calls in definition order (not execution order)
UC	A set of Used Classes. Elements of this birthmark is appeared in field type, method argument types, return type and used in methods.
CVFV	Constant values of field variables and its field type
k-gram	Construct k-gram from instructions.

We define the number of comparisons conducted between all the classes as "Comparison count", and the number of comparisons with similarity equal to or greater than 80% as "Filtered count". An average of "Filtered count" is final similarity (appeared Avg. in the following tables). Tables 7 - 11 show the results of bytecode comparison.

For obfuscated Android app, when compared to Section 4.1, we can see software birthmark comparison is much more effective. However, preservation of software birthmarks have some differences. The preservation of the birthmarks means the birthmark be preserved even if the original class file is tampered with. In the case of Opencamera app, preservation of the k-gram based birthmark was low. In the case of the Focal app and Connetbot app, UC birthmark showed a low preservation property.

Table 7: Opencamera_v.1.26 similarity between bytecodes using four birthmarks

	SMC		UC		CVFV		K-gram		
Features (options)	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Comparison count
Removal	100%	150	96%	421	100%	943	91%	119	23104
Renaming identifiers	100%	150	100%	26	100%	431	91%	117	23104
Byte code optimization	100%	150	96%	421	100%	816	91%	119	23104
String encryption	100%	150	96%	421	100%	831	92%	100	23104
Control flow obfuscation	100%	1125	97%	733	100%	1183	97%	329	29032
ALL	100%	1125	97%	733	100%	933	98%	314	29032

The results show Java birthmark schemes are more resilient than source code comparison scheme to obfuscation techniques. Obfuscation techniques make it difficult for Android app to be decompiled. However, they will change the appearance of bytecode generated from the same source. Source code comparison may detect that obfuscation has been applied, but may not overcome obfuscation techniques to measure similarity [25]. On the other hand, the results show that the birthmark schemes detect similarity two bytecodes before and after applying obfuscation techniques by at least 89% probability.

Table 8: Opencamera_v.1.27 similarity between bytecodes using four birthmarks

	SMC		UC		CVFV		K-gram		
Features (options)	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Comparison count
Removal	100.0%	150	96.2%	421	100.0%	937	91.6%	120	23104
Renaming identifiers	100.0%	150	96.2%	421	100.0%	432	91.7%	118	23104
Byte code optimization	100.0%	150	96.2%	421	100.0%	810	91.7%	120	23104
String encryption	100.0%	150	96.2%	421	100.0%	825	92.1%	101	23104
Control flow obfuscation	100.0%	1125	97.7%	694	100.0%	1177	97.8%	329	29032
ALL	100.0%	1125	97.7%	694	100.0%	933	98.1%	316	29032

Table 9: Focal_v.1.0 similarity between bytecodes using four birthmarks

	SMC		UC		CVFV		K-gram		
Features (options)	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Comparison count
Removal	99.9%	619	95.2%	1369	100.0%	1621	96.9%	406	59976
Renaming identifiers	99.9%	177	94.0%	487	100.0%	501	96.9%	116	20916
Byte code optimization	100.0%	706	95.4%	1532	100.0%	1426	97.2%	447	63504
String encryption	100.0%	706	95.4%	1532	100.0%	1426	96.9%	430	63504
Control flow obfuscation	100.0%	4886	97.2%	2577	100.0%	4341	99.4%	2234	87444
ALL	99.9%	177	94.0%	487	100.0%	528	96.7%	104	20916

5 Conclusion

Measuring software similarity between original programs and obfuscated ones is an approach to detect stolen software transformed by code obfuscator. In this paper, we analyze the effect of code obfuscation on the similarity of Android apps. We employ an off-the-shelf obfuscation tools to obfuscate open source Android apps. We measure the similarity between original apps and obfuscated apps, and analyze the effect of each obfuscation technique supported by the tool. The experiment results show that bytecode level similarity measure is more effective than source code level one.

Table 10: Connetbot similarity between bytecodes using four birthmarks

	SMC		UC		CVFV		K-gram		
Features (options)	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Comparison count
Removal	98.8%	927	91.7%	4967	99.9%	2540	94.4%	776	115432
Renaming identifiers	99.4%	1080	91.7%	6412	100.0%	2513	93.9%	1032	141376
Byte code optimization	99.4%	1079	91.7%	6412	100.0%	2892	94.0%	1033	141376
String encryption	99.4%	1079	91.7%	6412	100.0%	2889	94.7%	844	141376
Control flow obfuscation	100.0%	12500	94.8%	10300	100.0%	14070	99.1%	6270	232744
ALL	99.9%	10372	94.8%	8418	100.0%	11557	99.5%	4998	194392

Table 11: ShoppingList similarity between bytecodes using four birthmarks

	SMC		UC		CVFV		K-gram		
Features (options)	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Avg.	Filtered count	Comparison count
Removal	100.0%	5	90.9%	65	100.0%	126	91.6%	25	1720
Renaming identifiers	100.0%	215	94.9%	219	100.0%	529	97.9%	109	18017
Byte code optimization	100.0%	5	90.1%	67	100.0%	133	91.4%	25	1849
String encryption	100.0%	5	90.1%	67	100.0%	133	89.0%	18	1849
Control flow obfuscation	100.0%	100	92.3%	86	100.0%	209	95.6%	62	2666
ALL	100.0%	945	97.4%	377	100.0%	1058	99.5%	392	19608

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (NRF-2015R1D1A1A02061946).

References

[1] Google, “Google play stats,” http://www.appbrain.com/stats, [Online; Accessed on December 10, 2015].

[2] Arxan, “State of security in the app economy: Mobile apps under attack,” https://www.arxan.com/state-of-security-in-the-app-economy-research/, [Online; Accessed on December 10, 2015].

[3] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, “Detecting repackaged smartphone applications in third-party Android marketplaces,” in *Proc. of the 2nd ACM conference on Data and Application Security and Privacy (CODASPY’12), San Antonio, Texas, USA*. ACM, February 2012, pp. 317–326.

[4] J. Ko, H. Shim, D. Kim, Y.-S. Jeong, S.-j. Cho, M. Park, S. Han, and S. B. Kim, "Measuring similarity of Android applications via reversing and k-gram birthmarking," in *Proc. of the 2013 Research in Adaptive and Convergent Systems (RACS'13), Montreal, Quebec, Canada*. ACM, October 2013, pp. 336–341.

[5] Google, "Proguard," http://developer.android.com/tools/help/proguard.html, [Online; Accessed on December 10, 2015].

[6] T. Patki, "DashO Java obfuscator," http://www.cs.arizona.edu/~collberg/Teaching/620/2008/Assignments/tools/DashO/, [Online; Accessed on December 10, 2015].

[7] P. Solutions, "DashO - Java/Android enterprise protection and obfuscation," http://www.preemptive.com/products/dasho, [Online; Accessed on December 10, 2015].

[8] Licel, "Dexprotector," http://dexprotector.com, [Online; Accessed on December 10, 2015].

[9] "SourceForge.net," http://www.sourceforge.net/, [Online; Accessed on December 10, 2015].

[10] A. Aiken, "A system for detecting software plagiarism - MOSS," http://theory.stanford.edu/~aiken/moss/, [Online; Accessed on December 10, 2015].

[11] H. Tamada, "Stigmata," http://www.stigmata.osdn.jp/index.html, [Online; Accessed on December 10, 2015].

[12] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, August 2002.

[13] G. Naumovich and N. Memon, "Preventing piracy, reverse engineering, and tampering," *IEEE Computer*, vol. 36, no. 7, pp. 64–71, July 2003.

[14] G. Wroblewski, "General method of program code obfuscation (draft)," Ph.D. dissertation, Institute of Engineering Cybernetics, Wroclaw University of Technology, 2002.

[15] F. Buzatu, "Methods for obfuscating Java programs," *Journal of Mobile, Embedded and Distributed Systems*, vol. 4, no. 1, pp. 25–30, March 2012.

[16] D. Leskov, "Protect your Java code - through obfuscators and beyond," http://www.excelsior-usa.com/articles/java-obfuscators.html, October 2015, [Online; Accessed on December 10, 2015].

[17] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep. 148, 1997. [Online]. Available: https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf

[18] A. Jain, H. Gonzalez, and N. Stakhanova, "Enriching reverse engineering through visual exploration of Android binaries," in *Proc. of the 5th Program Protection and Reverse Engineering Workshop (PPREW'15), Los Angeles, California, USA*. ACM, December 2015. [Online]. Available: http://dx.doi.org/10.1145/2843859.2843866

[19] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto, "Design and evaluation of birthmarks for detecting theft of Java programs," in *Proc. of the 2004 IASTED International Conference on Software Engineering (IASTED SE'04), Innsbruck, Austria*. IAESTED, February 2004, pp. 569–574.

[20] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proc. of the 2005 ACM symposium on Applied Computing (ACM SAC'05), Santa Fe, New Mexico , USA*. ACM, March 2005, pp. 314–318.

[21] ——, "Detecting software theft via whole program path birthmarks," in *Proc. of the 7th International Conference on Information security (ISC'04), Palo Alto, California, USA, LNCS*, vol. 3225. Springer, Springer Berlin Heidelberg 2004, pp. 404–415. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30144-8_34

[22] G. Nolan, *Bulletproof Android: Practical Advice for Building Secure Apps*. Addison-Wesley Professional, 2014.

[23] A. Neshkov, "Androchef Java decompiler," http://www.neshkov.com/ac_decompiler.html, [Online; Accessed on December 10, 2015].

[24] K. W. Bowyer and L. O. Hall, "Experience using MOSS to detect cheating on programming assignments," in *Proc. in the 29th Annual Frontiers in Education Conference (FIE'99). San Juan, Pueto Rico*, vol. 3. IEEE, November 1999, pp. 13B3–18.

[25] B. S. Baker and U. Manber, "Deducing similarities in Java sources from bytecodes." in *Proc. of the 1998 USENIX Annual Technical Conference, New Orleans, Louisiana, USA*. USENIX, June 1998, pp. 179–190.

Author Biography

Jonghwa Park received the B.E. in Dept. of Software Science from Dankook University in 2015. He is currently a master student at Dept. of Computer Science and Engineering in Dankook University, Korea. His research interests include computer system security, mobile security, and software protection.

Hyojung Kim received the B.E. in Dept. of Computer Science from Dankook University in 2015. He is currently a master student at Dept. of Computer Science and Engineering in Dankook University, Korea. His research interests include computer system security and mobile security.

Younsik Jeong received the B.E. degree in computer engineering from Dankook University, Korea, in 2012 and the M.E. degree in computer science and engineering from the University of Dankook, Korea, in 2013. He is a Ph.D. student in Computer science and engineering at the Dankook University. His current research interests include computer security, mobile security.

Seong-je Cho received the B.E., the M.E. and the Ph.D. in Computer Engineering from Seoul National University in 1989, 1991 and 1996 respectively. He joined the faculty of Dankook University, Korea in 1997. He was a visiting scholar at Department of EECS, University of California, Irvine, USA in 2001, and at Department of Electrical and Computer Engineering, University of Cincinnati, USA in 2009 respectively. He is a Professor in Department of Computer Science and Engineering (Graduate school) and Department of Software Science (Undergraduate school), Dankook University, Korea. His current research interests include computer security, smartphone security, operating systems, and software protection.

Sangchul Han received his B.S. degree in Computer Science from Yonsei University in 1998. He received his M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 2000 and 2007, respectively. He is now an associate professor of Department of Computer Engineering at Konkuk University. His research interests include real-time scheduling, software protection, and computer security.

Minkyu Park received the B.E., M.E., and Ph.D. degree in Computer Engineering from Seoul National University in 1991, 1993, and 2005, respectively. He is now a professor in Konkuk University, Rep. of Korea. His research interests include operating systems, real-time scheduling, embedded software, computer system security, and HCI. He has authored and co-authored several journals and conference papers.