Open-Source Android App Detection considering the Effects of Code Obfuscation

Kyeonghwan Lim[1], Jungkyu Han[2], Byoung-chir Kim[1],
Seong-je Cho[1]*, Minkyu Park[3], and Sangchul Han[3]
[1]*Dankook University, Yongin, Republic of Korea*
{limkh120, gurukbc, sjcho}@dankook.ac.kr
[2]*NAVER Corp., Seongnam, Republic of Korea*
jungkyu.han@navercorp.com
[3]*Konkuk University, Chungju, Republic of Korea*
{minkyup, schan}@kku.ac.kr

Abstract

As open source software (Open Source Software, OSS) is becoming more and more popular, the risk of open-source license violation also increases. According to 2018 open source security and risk analysis report of Synopsys, 96% of applications (apps) include open source software and 74% of them them have licensing issues. To address this problem, many researchers have studied open-source licensing and OSS detection. However, most ones have conducted at source code level and have not considered the effects of code obfuscation. In this paper, we propose an effective technique to extract software birthmarks (i.e., features) from executable code of Android apps and find out whether the executable code is created from OSS by comparing the birthmarks of the executable code and those of known open-source apps. The proposed technique uses class hierarchy information (CHI) and control flow graphs (CFGs) as software birthmarks of Java bytecode code level. The CFG birthmark is robust against code obfuscation attacks and thus effective to detect open-source apps although their codes are obfuscated. We validate the proposed OSS detection technique through experiments on obfuscated apps.

Keywords: Open Source Software, similarity, control flow graph, class hierarchy information

1 Introduction

Many software companies develop software products using open-source software (OSS) for shortening the development period and saving the cost. OSS is widely used both as independent applications (apps) and as components in non-open-source apps. We can easily get many kinds of OSS from web hosting services such as GitHub, BitBucket, and F-Droid, etc [1, 2, 3]. OSS is a type of computer software released with its source code under a license where the copyright holder grants users the rights to reuse, change, and distribute the software under the license [4]. Examples of open-source licenses and free software license include GNU General Public License, GNU Lesser General Public License, BSD license, Apache License, Eclipse Public License, etc.

As the use of OSS has largely increased, license violations and security-related problems are also frequently occurring [5, 6]. According to Synopsys, an open source security management solution provider, '2018 Open Source Security and Risk Analysis' reports that 96% of applications include open source and

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), 9:3 (Sept. 2018), pp. 50-61
*Corresponding author: Department of Computer Science and Engineering, Dankook University, 152, Jukjeon-ro, Suji-gu, Yongin-si, Gyeonggi-do, Republic of Korea, Tel: +82-31-8005-3239

74% of them have licensing issues [5]. Some issues may be related to legal ones such as copyright infringement due to license violation. Taking advantage of OSS without violating the license is important. Thus, you should check if OSS you are using is under right license for your purpose.

Another problem is that your system can be hacked or your personal information can be leaked due to security vulnerabilities of OSS you are using. For example, Android provides useful functionalities to users using OSS such as OpenSSL and WebView. However, their vulnerabilities can bring about serious damage such as personal information hijacking [7]. If some vulnerabilities are exploited in very widely-used open-source projects such as Linux, MySQL, Apache, so on, the extent of damages becomes much wider.

We, therefore, need to check whether an app uses OSS modules without their license violation and which OSS it uses if any. It is also necessary to verify whether or not your software products contain well-known vulnerable OSS modules. In these cases, for software vendors, hardware vendors, and value-added-resellers, one of the most important things is to determine whether their proprietary and for-profit products use open-source frameworks, modules, components, and libraries. Then, if the products are created from or includes any OSS module/component/library, to detect and identify the OSS module/-component/library is necessary. This OSS detection is especially important to commercial apps because proprietary software arbitrarily made by open-source modules/components can cause license violation.

To address these problems, many studies have been conducted and many OSS detection tools are developed. Most of the developed tools have been implemented at source-code level [8, 9, 10] or using software birthmarks [11, 12, 13, 14, 15, 16]. A software birthmark is intrinsic characteristics of a program which can identify the program and detect software theft [17, 18]. In this paper, we also call the software birthmark the feature information of a program. The existing research efforts have limitations to handle obfuscated software. If software is obfuscated, its feature information may be changed. For examples, Google recommends using the code obfuscation tool, called Proguard [19], for protecting intellectual properties and business logics of software when developing Android apps, and most developers apply code obfuscations to their apps. Proguard performs obfuscation such as code optimizations; renaming of classes, methods and variables. As a result of the obfuscation, the features of the apps can be changed.

In this paper, we propose an effective technique to extract software birthmarks (i.e., features) from executable code of Android apps and find out whether the executable code is generated from OSS by comparing the birthmarks of the executable code and those of known open-source apps. The proposed technique uses class hierarchy information (CHI) [6] and control flow graphs (CFGs) as software birthmarks of Java bytecode code level. The CFG birthmark is robust against code obfuscation attacks and thus effective to detect open-source apps although their codes are obfuscated. We demonstrate the effectiveness of the proposed OSS detection technique through experiments on obfuscated apps. Our technique focus on detecting OSS at an Android app level not at a component/library level, or detecting obfuscated open-source Android app clone.

The rest of this paper is organized as follows. Section 2 discusses related works. Section 3 explains two software features for measuring similarity between two Android apps and proposes a new detection technique for open-source Android apps considering code obfuscation effects. In Section 4, we demonstrate effectiveness of the proposed technique and evaluate the performance. Conclusions and for future research are presented in Section 5.

2 Related Work

Most OSS detection schemes employ source code-based similarity analysis techniques. One of well-known OSS detection tools is Black Duck Software's Protex [10], which utilizes source codes and character strings to detect OSS. Other source code-based OSS detection tools include Clarity [8] and

Fossology [9]. These tools have difficulty in detecting OSS within the software whose source code is not available.

Some OSS detection tools utilize binary-level birthmarks if the source code of target software is not available. BAT [20] itself is an OSS under the Apache 2.0 license. It extracts symbol information and string table from binary codes to detect OSS. Kim et al. [16] proposed a function-level static birthmarking technique for detecting OSS libraries in Microsoft Windows Systems. Their approach extracted function parameter information and the size of local variables from binary and utilize them as birthmark. Extracting a relatively simple function-level birthmark and comparing them, the approach showed a good performance. However, because the size of local variables can vary with different compilers, it may not resilient to compiler optimization attacks. For example, a local variables can be easily removed by the compiler optimization. Thus, these binary-level OSS detection schemes are not effective if symbols are removed by compilation options or target binary is obfuscated. In order to remove the limitations in [16], Kim et al. devised a new birthmark which was more reliable than existing ones and robust to compiler optimization [21]. The newly devised birthmark is based on the attributes of function parameters such as the number, types, and order in Windows Systems. These attributes represent a unique property of a function and are resilient to compiler optimization. In [21], the birthmark uses the mapped types of function parameters of each function. That is, Kim et al. inferred parameter type(s) from a target binary and mapped a restored type onto one of newly defined three types on the basis of the memory size allocated to each type and the way of access to the memory. They extracted the birthmarks from target binary files and determined whether a binary file contained another binary (e.g., OSS component) by calculating the similarity between the extracted birthmarks. Their approach is efficient and effective to OSS components at binary level in Microsoft Windows Systems, however, it did not consider the effect of code obfuscation.

Becker et al. [6] proposed an obfuscation-resilient birthmark to detect third-party libraries within Android apps. They devised a light-weight tool *LibScout* that is resilient to common obfuscation schemes and capable of pinpointing exact library versions. In order to determine whether an app include a given library, *LibScout* uses high-level class organization information in class profile matching to get a similarity score between an app and a given library. Becker et al. utilized class hierarchy information (CHI) which is not influenced by control flow obfuscation nor API hiding. They build a database of CHI for third-party libraries, and the CHI extracted from target Android app is looked up in the database. However, the scheme is vulnerable to class renaming. In this paper, we propose an OSS detection scheme based on control flow graph (CFG) birthmark that is resilient to class renaming. Our scheme can improve the detection rate combined with CHI.

Zhang et al. [22] found that *LibScout* still has some limitations when applied to large-scale library detection. To solve the problems of *LibScout*, they proposed an obfuscation-resilient, highly precise and reliable library detector *LibPecker* for Android app. *LibPecker* adopts signature matching to get a similarity score between an app and a given library. By fully using the internal class dependencies inside a given library, it creates a strict signature for each class. *LibPecker* utilizes adaptive class similarity threshold and weighted class similarity score, then can tolerate library code optimization and elimination. While *LibPecker* introduces class dependency information and signature matching as software birthmarks, our technique employs CHI and CFG.

3 Detection Technique for Open-Source Android Apps

In this section, we propose an OSS app detection technique that is capable to detect OSS apps even for obfuscated Androids apps in execution code level. In other words, our technique does not need source codes of the Android apps. To explain concisely, we use two terms a “target app” and “queried OSS

apps". A target app is a suspicious app that is supposed to be generated from a certain open-source app. A target app can be an un-obfuscated app or obfuscated one. Queried OSS app (shortly, queried OSS) is the app known as an open-source app, that is, the app developed from OSS. Our technique determines if a target app is made from an open-source app by comparing the birthmark of the target app with that of each queried OSS app one by one. A software birthmark is a unique characteristic of an app which can identify the app and detect illegal software use [17, 18]. Software birthmarks have the same meaning as software features in this paper. Examples of software birthmarks include API information, control flow graph (CFG), call graph (CG), opcode n-gram, etc. For effectively determine whether a target app is generated from an open-source app, the proposed technique adopts two kinds of software birthmarks: class hierarchy information (CHI) [6] and control flow graphs (CFGs). Although CHI birthmarks shows robustness against some obfuscation methods such as API hiding [23, 24], control flow randomization [25, 26] and variable renaming [25, 26, 23, 24, 19], they can be ineffective to check out if a target app obfuscated by class renaming methods is built from an open-source app [6] because the CHI birthmarks in the obfuscated app can be modified (E.g., original class names can be transformed into obfuscated names).

Control flow graph (CFG) indicates a directed graph $G = (V, E)$ to represent all paths that might be traversed through a program during its execution. Each vertex $v \in V$ indicates a basic block that represents a maximal linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed). The instructions in the same basic block always executed in the same order and no outside instruction can execute between two instructions in the same block. Each directed edge $< v_i, v_j > \in E$ indicates that there may be a transfer of control from basic block v_i to v_j. Generally, one CFG is created for each method in an given Android app. Since CFG captures the structural information such as execution orders between instructions and control transfer in target apps, CFG is able to detect some patterns originated from the query OSS even in the target apps obfuscated by class renaming.

The proposed detection technique is roughly divided into the two phases : (1) Reference birthmark construction phase and (2) OSS detection phase. In the reference birthmark construction phase, birthmarks of open-source apps are extracted and maintained in a database as reference birthmarks. The reference birthmarks are then used in OSS detection phase, where we can judge whether a target app is identical or similar to a queried OSS by comparing the birthmark extracted from the target app with the reference birthmark one by one.

Reference birthmark construction phase: each source code is downloaded one after another from an open-source Android app repository such as F-Droid [3] and then compiled into an executable file format. CHI and CFG birthmarks are extracted from the compiled apps. We use the procedure [6] for extracting CHI and *dexdump* [27] tool for extracting CFG respectively. The extracted birthmarks are stored in a database as the reference birthmarks representing queried OSSs.

OSS detection phase: Figure 1 shows OSS detection process of our proposed technique. When a target app is given, CHI and CFG birthmarks are extracted from the target app in a similar way to extracting the reference birthmark (Extracting Features). Then the CHI birthmark of the target app and each of reference birthmarks are compared (Comparing Merkle trees). If CHI comparison is failed because a class renaming obfuscation technique is applied to the target app, a subgraph similarity comparison between the CFGs of the target app and the CFGs of each queried OSS app is carried out to judge whether the target app is identical or similar to one of queried OSS apps (Computing subgraph isomorphism and Measuring similarity of node).

The similarity between a CHI birthmark of the target app and that of a queried OSS is first evaluated by using Eq.(1).

$$sim(tgt, query) = \frac{|C_{tgt} \cap C_{query}|}{|C_{query}|} \in [0, 1] \quad (1)$$

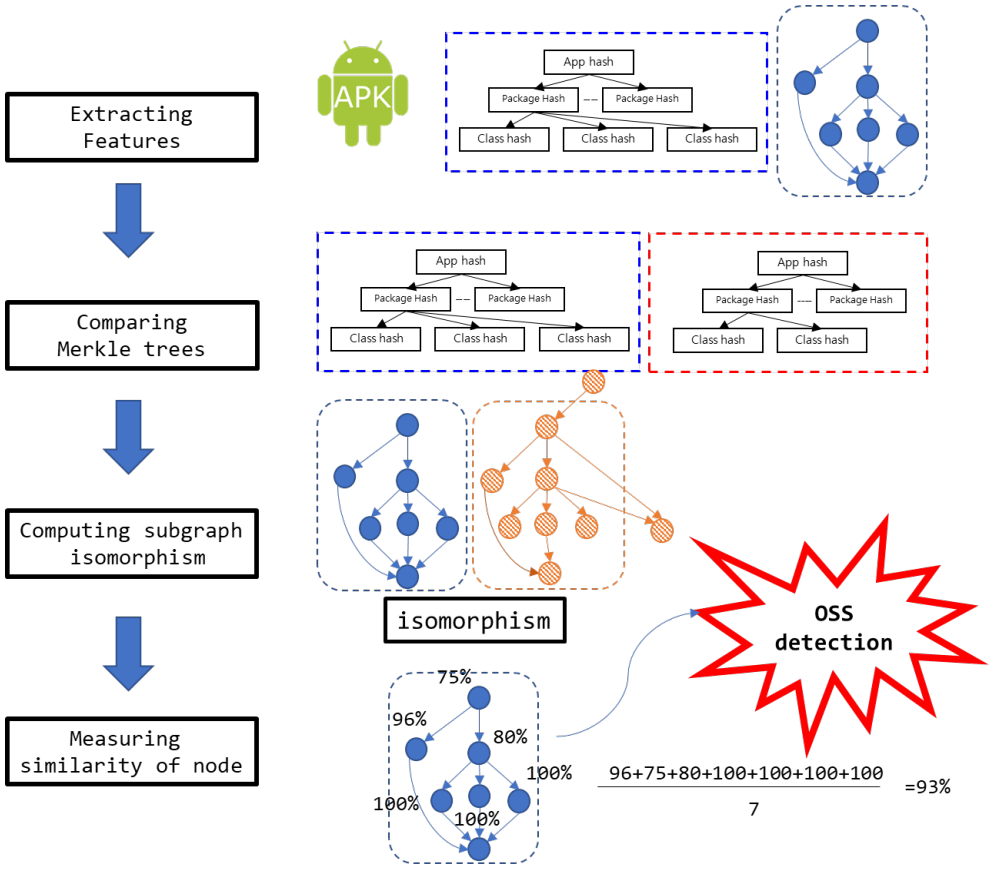

Figure 1: OSS detection phase

where, C_x indicates the set of classes in the CHI of App x. tgt and $query$ indicate the target app and queried OSS respectively. When the value of $sim(tgt, query)$ is more than empirically predefined threshold $\theta_{CHI} = 0.7$, we consider the target app is identical or similar to the quired OSS.

The similarity degree of CHI birthmarks is small and the CHI comparison can fail when a target app is obfuscated by class renaming. In such cases, our technique carries out the comparison between CFG birthmarks using subgraph isomorphism. Subgraph isomorphism detection is adopted to judge a given CFG is a sub-graph of the other CFG. Figure 2 shows an example of subgraph isomorphism. Since a sub-graph of graph G2 in the dashed square has the same topology to graph G1, G1 and G2 are in a relationship of subgraph isomorphism. We used VF2 algorithm [28, 29] to detect subgraph isomorphism.

Since the subgraph isomorphism problem is a NP-complete problem, to solve the problem in polynomial time, we skip the comparison of two CFGs when the ratio of the orders of small graph to big graph (Eq.(2)) is small. More specifically we calculate $r(g_1, g_2)$ with all possible CFG pairs, and skip the comparison for the graph pairs that $r(g_1, g_2)$ is smaller than predefined threshold θ_{CFG}. The appropriate value of θ_{CFG} is evaluated in Section 4.2.

$$r(g_1, g_2) = \frac{\min(Order(g_1), Order(g_2))}{\max(Order(g_1), Order(g_2))} \in [0, 1] \tag{2}$$

where g_x indicates a CFG, $Order(g_x)$ indicates the number of vertices of g_x. For instance, if $r(g_{ref1}, g_{tgt}) = 0.3$, $r(g_{ref2}, g_{tgt}) = 0.8$ and $\theta_{CFG} = 0.5$ then we discard the subgraph isomorphism detection for $(g_{ref1},$

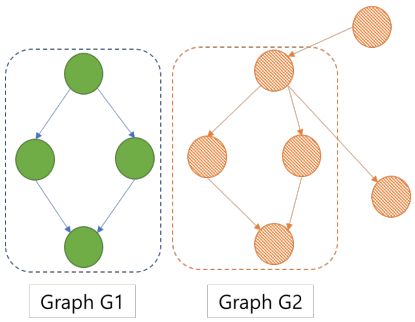

Figure 2: Subgraph isomorphism

g_{tgt}) pair while the detection is carried out for (g_{ref2}, g_{tgt}) pair.

Even we detect the existence of subgraph isomorphism between two CFGs, there is a possibility that the detected isomorphism is caused by chance. To reduce false detection possibility, similarities between two basic blocks that located in the same topological position (vertex) in the detected sub-CFG are calculated and if the average similarity of all compared basic block pairs exceeds predefined threshold $\theta_{CB} = 0.7$, we consider the two CFGs are similar. We employ an instruction order similarity of two basic blocks. The similarity between two basic block b_1 and b_2 follows Eq.(3).

$$sim(b_{tgt}, b_{query}) = \frac{2 \cdot Len(LCS(b_{tgt}, b_{query}))}{Len(b_{tgt}) + Len(b_{query})} \in [0,1] \tag{3}$$

where, b_{tgt} and b_{query} indicate the instruction sequence in a basic block of the target CFG and that of the queried CFG respectively. $Len(x)$ indicates the length of sequence x. $LCS(b_{tgt}, b_{query})$ indicates Longest Common Sebsequence (LCS) between b_{tgt} and b_{query}. For instance, the similarity between the two basic blocks shown in Figure 3 is 0.705 ($= 2 \cdot 6/(9+8)$) and the two sub-graph shown in Figure 4 is similar because the average of the similarities of two basic blocks in the same topological position exceeds $\theta_{CB} = 0.7$.

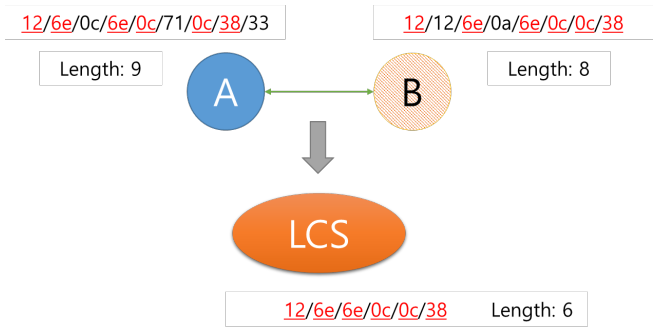

Figure 3: LCS between two basic blocks

4 Evaluation

In this section, we show the effectiveness of our proposed technique by conducting experiments with open-source apps. We first describe the open-source apps used in the evaluation, and then explain the result of evaluation.

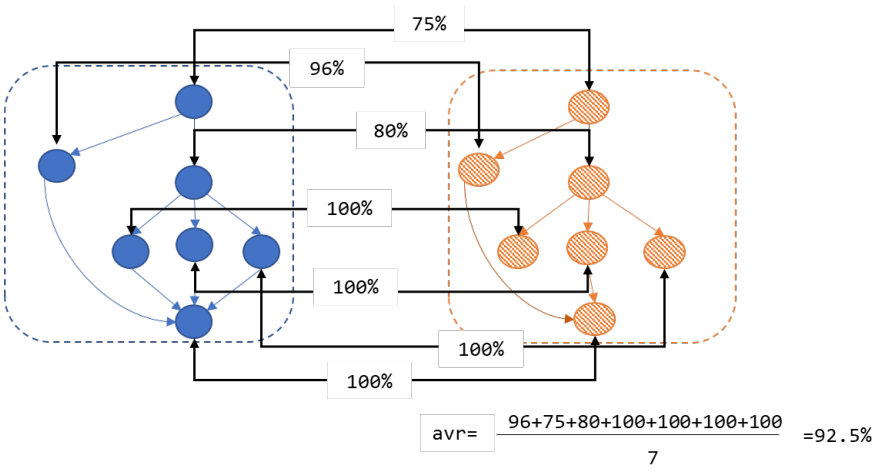

Figure 4: Similarity between two sub-CFGs in graph isomorphism

4.1 Dataset for Reference birthmarks

We prepared two groups of open-source apps for the evaluation. The first group consists of original open-source apps that any obfuscation methods have not been applied to. It contains 91 open-source apps collected from the open-source repository F-Droid [3]. The second group consists of obfuscated open-source apps which have been transformed from the apps in the first group using ProGuard [1]. The second group includes 91 obfuscated open-source apps. In this paper, we call the first group the 'intact OSS App group,' and the second group the 'obfuscated OSS App group.' In this paper, each open-source app is a file with an executable file format (i.e., an APK file including DEX) for Android smartphones. The open-source apps are compiled and built from source programs in Java programming language. The CHI and CFG birthmarks are extracted from both the intact OSS App group and the obfuscated OSS App group. The CHI and CFG birthmarks of intact OSS App group are stored in the database as reference birthmarks. The CHI or CFG birthmarks of an obfuscated app can be different from each other depending on code obfuscation tools and obfuscation techniques. Therefore, the birthmarks of obfuscated OSS App group are not maintained as the reference birthmarks. See the reference [30] for various code obfuscation tools and obfuscation techniques.

ProGuard is one of the most popular obfuscation tools for Android Apps. ProGuard transforms the names of classes, methods, and variables, and then removes unnecessary/dummy codes. However, the names of some frequently used methods and classes are not obfuscated by ProGuard in order to prevent the degradation of execution speed as well as introduction of errors. It is possible to tell ProGurard not to obfuscate certain classes and methods. Figure 5 shows the default list of classes and methods which Proguard does not obfuscate. By default, all classes and methods shown in Figure 5 are preserved without being obfuscated.

4.2 Detection performance

We first evaluates the performance of CHI birthmark comparison (Comparing Merkle trees in Figure 1). As shown in Table 1,all apps in the intact OSS App group are detected (100% detection ratio) while only 58 apps of total 91 apps in the obfuscated OSS group are detected (64% detection ratio). Figure 6 shows the package structure of an obfuscated app that is not detected by CHI birthmark comparison. Manual

[1] https://www.guardsquare.com/en/products/proguard

```
-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
-keep public class * extends android.content.BroadcastReceiver
-keep public class * extends android.content.ContentProvider
-keep public class * extends android.app.backup.BackupAgentHelper
-keep public class * extends android.preference.Preference
-keep public class com.android.vending.licensing.ILicensingService
-keep public class acr.browser.lightning.reading.*
-keep class org.lucasr.twowayview.** { *; }

-keepclasseswithmembernames class * {
    native <methods>;}
-keepclasseswithmembernames class * {
    public <init>(android.content.Context, android.util.AttributeSet);}
-keepclasseswithmembernames class * {
    public <init>(android.content.Context, android.util.AttributeSet, int);}
```

Figure 5: The ranges of classes and methods excluded from obfuscation

inspection revealed that the class names of 33 undetected Apps were obfuscated while the class names of 58 detected Apps remained intact even after obfuscation.

Table 1: CHI birthmark comparison results

	Intact OSS Apps (Detected/All/Ratio)	Obfuscated OSS Apps (Detected/All/Ratio)
Detection rate	91 / 91 / 100%	58 / 91 / 64%

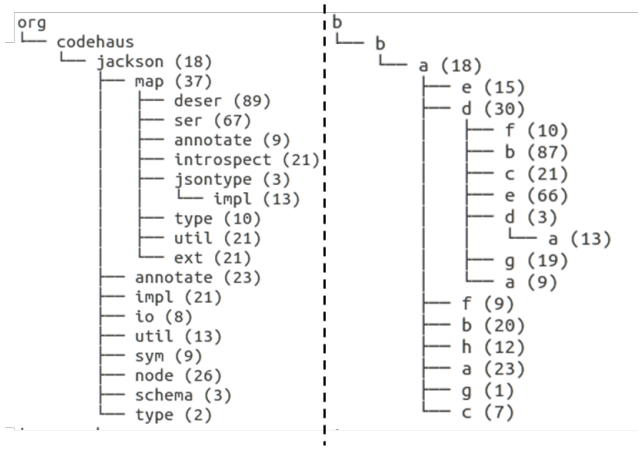

Figure 6: The package structure of an obfuscated APP (Original (Left), Obfuscated (Right))

For detecting the obfuscated apps that were not detected by CHI birthmark comparison, the CFG subgraph similarity comparisons are carried out (Computing subgraph isomorphism and Measuring similarity of node in Figure 1). We changed the value of θ_{CFG} from 1.0 to 0.6 by decreasing the value by 0.1 and evaluated the obfuscated OSS App detection ratio. Table 2 shows that the detection ratio is improved as the value of θ_{CFG} decreases and 76 out of 91 OSS Apps are detected when $\theta_{CFG} = 0.6$. One possi-

ble reason of 15 App detection failures is that too many or a few control blocks reside in CFGs. Since the subgraph isomorphism problem is a NP-complete problem, VF2 algorithm [28, 29] fails to compare CFGs in polynomial time when there exists many control blocks in CFGs. When an App has too simple control flows, OSS App detection also fails because there exists insufficient number of CFGs to compare.

Table 2: CFG birthmark comparison results for obfuscated OSS Apps

θ_{CFG}	Additional Detection ratio (Detected/All/Ratio)	Total Detection ratio (Detected/All/Ratio)
1.0	8 / 33 / 24.24%	66 / 91 / 72.53%
0.9	9 / 33 / 27.27%	67 / 91 / 73.63%
0.8	14 / 33 / 42.42%	72 / 91 / 79.12%
0.7	15 / 33 / 45.45%	73 / 91 / 80.22%
0.6	18 / 33 / 54.54%	76 / 91 / 83.52%

5 Conclusion and Future Work

In this paper, we have proposed an effective technique that is capable to detect open-source Android apps even though the apps have been obfuscated by ProGuard, which is the most popular obfuscator for Java bytecode. Our technique operates at the executable code level and does not require source code. The technique has adopted two software birthmarks, class hierarchy information (CHI) and control flow graph (CFG), where the CFG birthmark has showed substantial improvements for detecting the open-source apps obfuscated by class renaming scheme. Using the CHI and CFG birthmarks together, the detection ratio is improved from 24% to 54% compared to the case of only using the CHI birthmark. Although LibScout [6] proposed by Backes et al. got some resilience against some obfuscation schemes, there are still some limitations such as too relaxed class profiling, and difficulty in detecting OSS obfuscated with class renaming scheme. In this paper, we have solved one of the problems of LibScout using the CFG birthmark. Our technique can mitigate the workload of painstaking tasks to manually detect an open-source app in online marketplaces for Android apps.

Since more than 54% of detection ratio improvement is restricted by computational cost of subgraph isomorphism detection, devising an efficient approximation algorithm to detect subgraph isomorphism is one of our future works. In addition, we plan to extend our technique to detect open-source components/libraries or Java methods at execution code level.

Acknowledgments

This research was supported by (1) Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(no. NRF-2015R1D1A1A02061946) and (2) the MIST(Ministry of Science and ICT), Korea, under the National Program for Excellence in SW supervised by the IITP(Institute for Information & communications Technology Promotion)(2017-0-00091)

References

[1] GitHub, "Github," https://github.com [Online; accessed on August 19, 2018].

[2] bitbucket, "Bitbucket," https://bitbucket.org [Online; accessed on August 19, 2018].

[3] F-Droid, "F-droid," https://f-droid.org/en/ [Online; accessed on August 19, 2018].

[4] Wikipedia, "Open-source software," https://en.wikipedia.org/wiki/Open-source_software [Online; accessed on August 19, 2018].

[5] Synopsys, "2018 open source security and risk analysis," https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2018-ossra.pdf [Online; accessed on August 19, 2018], May 2018.

[6] M. Backes, S. Bugiel, and E. Derr, "Reliabl third-party library detection in android and its security applications," in *Proc. of the 23rd ACM Conference on Computer and Communications Security (CCS'16), Hofburg Palace, Vienna, Austria*. ACM, October 2016, pp. 356–367.

[7] M. Neugschwandtner, M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A view to a kill: Webview exploitation," in *Proc. of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET'13), Washington, D.C*. USENIX, November 2013.

[8] clarityinsights, "Clarity," https://www.clarityinsights.com [Online; accessed on August 19, 2018].

[9] fossology, "Fossology," https://www.fossology.org/ [Online; accessed on August 19, 2018].

[10] synopsys, "Protex," https://www.blackducksoftware.com/products/protex [Online; accessed on August 19, 2018].

[11] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discover: Efficient cross-architecture identification of bugs in binary code," in *Proc. of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16), San Diego, California, USA*. The Internet Society, February 2016.

[12] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph based bug search for firmware images," in *Proc. of the 23rd ACM Conference on Computer and Communications Security (CCS'16), Hofburg Palace, Vienna, Austria*. ACM, October 2016, pp. 480–491.

[13] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Proc. of the 10th International Conference on Information and Communications Security (ICICS'08), Birmingham, UK*, ser. Lecture Notes in Computer Science, vol. 5308. Springer-Verlag, October 2008, pp. 238–255.

[14] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, January 2017.

[15] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proc. of the 8th Working Conference on Mining Software Repositories (MSR'11), Waikiki, Honolulu, Hawaii, USA*. ACM, May 2011, pp. 63–72.

[16] D. Kim, S. Cho, S. Han, M. Park, and I. You, "Open source software detection using function-level static software birthmark," *Journal of Internet Services and Information Security*, vol. 4, no. 4, pp. 25–37, November 2014.

[17] J. Choi, Y. Han, S.-j. Cho, H. Yoo, J. Woo, M. Park, Y. Song, and L. Chung, "A static birthmark for ms windows applications using import address table," in *Proc. of the 2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS'13), Taichung, Taiwan*. IEEE, July 2013, pp. 129–134.

[18] 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 onvergent Systems (RACS'13), Montreal, Quebec, Canada*. ACM, October 2013, pp. 336–341.

[19] guardsquare, "Proguard," https://www.guardsquare.com/en/proguard [Online; accessed on August 19, 2018].

[20] binaryanalysis, "Binary analysis tool," http://www.binaryanalysis.org/en/home [Online; accessed on August 19, 2018].

[21] D. Kim, S.-j. Cho, M. Park, and S. Han, "Open source software detection using function parameter based software birthmark," *Journal of Internet Technology*, vol. 18, no. 4, pp. 801–811, July 2017.

[22] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *Proc. of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'18), Campobasso, Italy*. IEEE, March 2018, pp. 141–152.

[23] guardsquare, "Dexguard android obfuscator," https://www.guardsquare.com/en/dexguard [Online; accessed on August 19, 2018].

[24] dexprotector, "Dexprotector android obfuscator," https://dexprotector.com [Online; accessed on August 19, 2018].

[25] allatori, "Allatori java obfuscator," http://www.allatori.com [Online; accessed on August 19, 2018].

[26] preemptive, "Dasho java obfuscator," https://www.preemptive.com/products/dasho/overview [Online; accessed on August 19, 2018].

[27] Y. Wei, "Dexdump," https://github.com/greatyao/dexdump [Online; accessed on August 19, 2018].

[28] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," in *Proc. of the 3rd IAPR TC-15 Workshop on Graphbased Representations in Pattern Recognition (GbRPR'01), Ischia, Italy*. CUEN, May 2001, pp. 149–159.

[29] P. Foggia, C. Sansone, and M. Vento, "A performance comparison of five algorithms for graph isomorphism," in *Proc. of the 3rd IAPR TC-15 Workshop on Graphbased Representations in Pattern Recognition (GbRPR'01), Ischia, Italy*. CUEN, May 2001, pp. 188–199.

[30] P. Faruki, H. Fereidooni, V. Laxmi, M. Conti, and M. Gaur, "Android code protection via obfuscation techniques: Past, present and future directions," *arXiv:1611.10231*, november 2016.

Author Biography

Kyeonghwan Lim received the B.E. degree in Dept. of Software Science from Dankook University, Korea, in 2015 and the M.E. degree in computer science and engineering from Dankook University, Korea, in 2016. He is currently a Ph.D. student in Computer Science and Engineering at Dankook University, Korea. His research interests include computer system security and mobile security.

Jungkyu Han received the B.E. and M.E. degrees in Computer Science and Engineering from Seoul National University in 2005 and 2007 and received Ph.D degree in Computer science and Communications Engineering from Waseda University in 2018. He worked for NTT in Japan from 2007 to 2014. Now he works for NAVER corp. in Korea. His research interests include data mining, artificial intelligence, distributed computing and computer security.

Byoung-chir Kim is currently an undergraduate student at Dept. of Software Science in Dankook University, Korea. His research interests include computer system security, mobile security.

Seong-je Cho received the B.E., M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 1989, 1991 and 1996, respectively. In 1997, he joined the faculty of Dankook University, Korea, where he is currently a Professor in Department of Computer Science and Engineering (Graduate school) and Department of Software Science (Undergraduate school). He was a visiting research professor 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. His current research interests include computer security, mobile app security, operating systems, and software intellectual property protection.

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.

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 Dept. of Software Technology at Konkuk University. His research interests include real-time scheduling, and computer security.