A Framework for Identifying Obfuscation Techniques applied to Android Apps using Machine Learning

Minjae Park[1], Geunha You[1], Seong-je Cho[1], Minkyu Park[2] and Sangchul Han[2*]
[1]*Dankook University, Yongin, Korea*
{parkminjae, geunhayou, sjcho}@dankook.ac.kr
[2]*Konkuk University, Chungju, Korea*
{minkyup, schan}@kku.ac.kr

Received: November 1, 2019; Accepted: December 7, 2019; Published: December 31, 2019

Abstract

Malicious app writers tend to employ code obfuscation techniques to prevent their malicious code from being easily reverse engineered and analyzed. In order to effectively analyze malicious Android apps, it is necessary to identify what code obfuscation technique is applied to the malicious apps. Existing studies have devised some approaches that identify app-level obfuscation. However, recent obfuscators can apply different obfuscation techniques on a class-by-class basis not on an app basis. In such a case, app-level obfuscation identification may be ineffective. In this paper, we propose a new framework to identify a class-level obfuscation technique used in Android apps. The proposed framework vectorizes the decompiled codes of each class of Android apps using a paragraph vector. Then the output vectors are fed to machine learning classifier to identify what obfuscation technique is applied to each class. We use four machine learning classifiers: Random Forest, AdaBoost, Extra Trees, and Linear SVM, and compare the performance of the classifiers for each obfuscation technique.

Keywords: Android app, Obfuscation technique, Class-level obfuscation, Machine learning

1 Introduction

Since Android applications (apps) are easy to decompile or disassemble, it is easy to reverse engineer and analyze the codes of Android apps. Android app developers apply obfuscation techniques to their apps to protect their intellectual property, business logics, algorithms or security-significant codes from code theft. Well-known code obfuscation tools for Android apps are ProGuard [7], dashO [2], DexProtector [4] and DexGuard [3].

Malware writers also employ code obfuscation techniques to deceive anti-malware or play store. Code obfuscation makes it difficult to analyze codes or find malware signature in the codes. If the obfuscation techniques applied to a given Android malware can be identified, the analysis or de-obfuscation of the malware can be performed efficiently.

Android obfuscation techniques include identifier renaming, string encryption, control flow obfuscation, reflection obfuscation, etc. Identifier renaming alters the structure of source and binary. It usually changes the names of packages, classes, methods and fields. A techniques that changes the package hierarchy is a sort of identifier renaming technique. String encryption techniques encode character strings a priori, and decode them at runtime. Control flow obfuscation techniques modify the control flow of the

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), 10(4):22-30, Dec. 2019
DOI:*10.22667/JOWUA.2019.12.31.022*

*Corresponding author: Department of Software Technology, Konkuk University, 268 Chungwondaero, Chungju-si, Chungcheongbuk-do, 27478, Korea, Tel: +82-43-840-3605

codes by inserting dummy codes, extending loop condition, inserting try-catch clauses. Reflection obfuscation techniques utilize reflection methods in order to make the method invocation analysis difficult.

There are research works on the identification of app-level obfuscation. Wermke et al. [20] proposed a signature-based tool OBFUSCAN that can identify the obfuscation options of Proguard. Wang and Rountev [19] presented a machine learning-based two-stage technique that can identify obfuscators and their configuration. Dong et al. [11] proposed machine learning-based or signature-based detection models for four obfuscation techniques. Mirzaei et al. [17] also proposed a machine learning-based (SVM) obfuscation technique detector named AndrODet.

Many obfuscators support class-level configuration, i.e., different obfuscation techniques can be applied on a class-by-class basis, or only a subset of classes is selected for obfuscation. In this case, the results of app-level obfuscation technique identification may be inaccurate. Hence, it is necessary to study class-level obfuscation technique identification. If we can identify the applied obfuscation techniques for each class, we can save the resources for app analysis or employ proper analysis techniques for each class.

In this paper, we present a novel framework for class-level obfuscation technique identification. The framework vectorizes the decompiled codes of each class of Android apps using Paragraph Vector [16], a code vectorization technique that learns fixed-length feature representations from variable-length texts. Then the output vectors are fed to machine learning classifiers to identify what obfuscation technique is applied to the class. We adopt four machine learning algorithms: Random Forest [9], AdaBoost [13], Extra Trees [15] and Linear SVM [10]. We compare the performance of the classifiers for each obfuscation technique.

This paper is organized as follows. Section 2 explains the background knowledge about code obfuscation and related studies. Section 3 proposes the framework for class-level obfuscation technique identification. Section 4 demonstrates the experiments and results. Section 5 concludes our study and presents future work.

2 Background and Related Work

Code obfuscation is a technique of changing the physical appearance of code while preserving its original semantics and functionalities [12]. Code obfuscation is an efficient way to protect software from reverse engineering. Android app obfuscation techniques include identifier renaming, string encryption, control flow obfuscation, and reflection obfuscation. Identifier renaming is a type of alteration of the program's code and binary structure, usually by changing package, class, method, and field names. This also includes techniques for changing the package hierarchy. When a user applies an identifier renaming technique, he or she may also formulate the renaming policy manually. However, most users use the default policy provided by their obfuscator. So we consider the default policies of obfuscators in this work. String encryption is a technique that encrypts a string constant, stores it and decrypts it at runtime to output the original string. Control flow obfuscation is a technique that makes it difficult to analyze codes by changing their control flow. It inserts dummy code, changes loops, and inserts try-catch statements. For example, inserting a try-catch statement makes it difficult to know whether to execute a try block or a catch block at runtime. Adding/changing a switch or if statement makes the code difficult to be analyzed. Reflection obfuscation is a technique that makes it difficult to analyze method calls using reflection methods. You can make it difficult to find a method that is actually called by replacing a method call with a call through reflection method.

Well-known Android app obfuscators are ProGuard, dashO, DexProtector, and DexGuard. ProGuard is an open source obfuscation tool that provides features such as code reduction, optimization, and obfuscation [7]. dashO is a commercial obfuscation tool that provides functions such as identifier renaming,

string encryption, and control flow obfuscation [2]. DexProtector is another commercial obfuscation tool. It provides functions such as string encryption, class encryption, and resource encryption [4]. DexGuard is also a commercial obfuscation tool that provides features such as identifier renaming, string encryption, control flow obfuscation, reflection obfuscation, asset encryption, and class encryption [3].

There are many research works on signature-based or machine learning-based obfuscation detection. Wermke et al. [20] proposed a signature-based obfuscation detection method and a tool that identifies the obfuscation techniques of ProGuard. By mimicking the obfuscation procedure of ProGuard, they detected the identifier renaming and analyzed the distribution of obfuscated method names to create signature. Wang and Rountev [19] proposed an obfuscation detection method for Android apps using machine learning (SVM). The proposed method consists of two steps. The first step identifies the obfuscation tool based on the names of files, packages, classes, methods, fields, and external package objects. The second step identifies the options of the identified obfuscation tool. The experiments were performed using open source or free version of commercial obfuscation tools. Dong et al. [11] also proposed an obfuscation detection method for Android app using machine learning (SVM). They used fixed-length feature vectors generated from N-grams of class, method, and field names for identifier renaming detection. The feature vector for string encryption detection was generated from N-grams of string constants. They use the sequence pattern of the reflection method as a feature vector for reflection identification. Mirzaei et al. [17] also proposed an SVM-based obfuscation detection method for Android apps. The feature vector for identifier renaming detection includes the average length of words, the average distance of consecutively extracted identifiers, etc. For the feature vector for string encryption detection, the average entropy, the average length of constant strings, etc. were used. The feature information for the control flow obfuscation includes the number of nodes, the number of edges from each application's control flow graph, the number of goto instructions per code line, and the number of NOP instructions per code line. Suarez-Tangil et al. [18] proposed a DT-based obfuscation detection method for Android apps. They designed a fast and accurate classifier against obfuscated Android malware using obfuscation-invariant features and artifacts introduced by obfuscation mechanisms used in malware. Garcia et al. [14] proposed an obfuscation-resilient Android malware detection and family identification method based on machine learning (DT and KNN). It contains four types of features based on Android API: Sensitive API Extractor, Information Flow Extractor, Intent Action Extractor and Package API Extractor.

However, signature-based or machine learning approach through feature engineering might have a difficulty when other versions of the tool or customized obfuscators are used. In addition, some obfuscators can apply obfuscation techniques at class-level or method-level rather than at app-level. Even if an analyst identifies obfuscation options applied at app-level, he or she must accurately identify again which classes or methods have been obfuscated.

3 Class-Level Obfuscation Technique Identification for Android Apps

3.1 Framework Overview

This section presents a novel framework for class-level obfuscation technique identification. Figure 3.1 illustrates the overview of the framework. In the first step (Preprocessing), an APK file is decompiled using AndroGuard [1], an Android app analysis tool. It extracts classes from classes.dex in the APK file and generates Java codes for each class. In the second step (Code embedding), vectors are generated from Java class codes using Paragraph Vector [16]. Paragraph Vector is an algorithm that learns fixed-length feature representations from variable-length texts. Paragraph Vector has two models: PV-DM (Paragraph Vector – Distributed Memory) and PV-DBOW (Paragraph Vector – Distributed Bag of Words). PV-DM considers the concatenation of paragraph vectors and word vectors to predict the next word in a context. PV-DBOW predicts randomly sampled words from paragraphs. In our work, we combine PV-DM and

PV-DBOW as recommended in [16], and implement the combined model using gensim [6]. In the third step (Training), the code vectors generated in the previous step are fed to machine learning classifiers for training. In the last step, the framework tests and identifies the obfuscation technique applied to given code vectors. The employed classifiers are Random Forest [9], AdaBoost [13], Extra Trees [15] and Linear SVM [10]. They are implemented using scikit-learn [8]. The framework was implemented using Python 2 and the experiments were performed on Ubuntu 16.04.4.

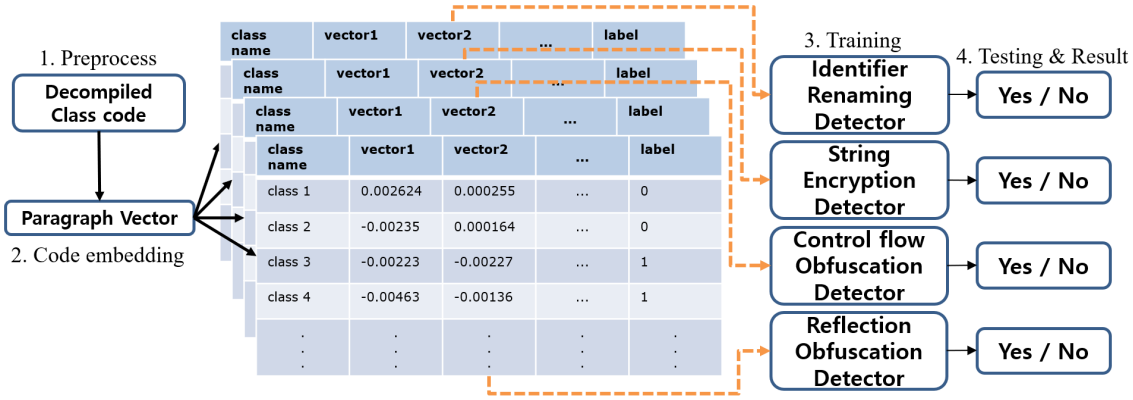

Figure 1: Schematic Diagram of Proposed Framework

3.2 Dataset

We collected 148 unobfuscated Android apps from F-Droid [5]. We refer to these apps as the original apps. We created 1,717 obfuscated apps by applying ProGuard, dashO, DexProtector and DexGuard. The applied obfuscation techniques are identifier renaming, string encryption, control flow obfuscation and reflection obfuscation. Apps that failed to be either obfuscated or decompiled were excluded from the dataset. The total number of apps in the dataset is 1,865 (=148+1,717).

For our experiments, we generated obfuscated apps varying detailed options of obfuscators as follows. With ProGuard default or default-optimize option for identifier renaming was applied. With dashO a combination of identifier renaming options (Overload-Induction or Simple), string encryption options (Level 1, 5 or 10) and control flow obfuscation options (MaxNumber 1, 5 or 10) was applied. With DexGurad Low, Normal or High option for control flow obfuscation was applied. There is no detailed option for identifier renaming, string encryption and reflection obfuscation in DexGuard. With DexProtector default option for string encryption was applied. Table 1 shows the number of apps and their classes generated by each obfuscator. The proposed framework stores each decompiled class as a separate file. The total number of decompiled class files is 151,391. They are randomly divided into a training set (80%) and a testing set (20%) for our experiments.

4 Experiments

The proposed framework vectorizes the Java codes of the decompiled classes. These vectors are used as features in machine learning classifiers where a true positive (TP) is a prediction of application of an obfuscation technique that turns out to be correct. The employed machine learning algorithms are Random Forest, AdaBoost, Extra Trees and Linear SVM. The metrics of the experiments are accuracy,

Table 1: The number of obfuscated apps and classes

		# of apps	# of classes				
			total	identifier renaming	string encryption	control flow obfuscation	reflection obfuscation
original		148	15,733	·	·	·	·
obfuscated & decompiled	ProGuard	182	11,443	11,443	·	·	·
	DashO	912	84,501	21,150	31,660	31,691	·
	DexGuard	489	23,842	4,572	5,025	9,657	4,558
	DexProtector	134	15,872	·	15,872	·	·
total		1,865	151,391	37,165	52,557	41,348	4,558

precision, recall and F1-score, and their definitions are as follows.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}, \qquad Precision = \frac{TP}{TP+FP},$$

$$Recall = \frac{TP}{TP+FN}, \qquad F1-score = \frac{2\times(Precision\times Recall)}{Precision+Recall}$$

We perform a grid search to investigate the best candidate parameters using scikit-learn. In the experiments with Random Forest, AdaBoost and Extra Trees, we vary the number of trees (a parameter that controls the number of decision tress) and max_depth (a parameter that controls the maximum depth of Trees) between 10 and 100 with step of 10, respectively. In the experiments with Linear SVM, we vary gamma (a parameter that controls the radius of influence of samples) and C (a parameter that controls the hardness of margin) as 0.0001, 0.001, 0.01, 0.1, 1, 10 and 100.

Table 2: Experiment results with Random Forest

	Accuracy	Precision	Recall	F1-score	training time	# of trees	max_depth
identifier renaming	0.698	0.487	0.698	0.573	9.181s	40	10
string encryption	0.769	0.591	0.769	0.669	16.188s	30	20
control flow obfuscation	0.755	0.743	0.755	0.705	12.984s	20	100
reflection obfuscation	0.792	0.627	0.792	0.700	16.78s	90	40
average	0.754	0.612	0.754	0.662	13.783s	·	·

Table 2, 3, 4 and 5 show the experiment results with Random Forest, AdaBoost, Extra Trees and Linear SVM, respectively. The tables present the parameters with which each classifier shows the best performance in each obfuscation technique detection. The performances of the four algorithms are roughly similar. In identifier renaming detection, string encryption detection and control flow obfuscation detection, Extra Trees performs slightly better than other algorithms with accuracy 0.701, 0.770 and 0.800, respectively. In reflection obfuscation detection, Random Forest outperforms other algorithms with accuracy 0.792.

Table 3: Experiment results with AdaBoost

	Accuracy	Precision	Recall	F1-score	training time	# of trees	max_depth
identifier renaming	0.688	0.585	0.688	0.59	697.704s	100	50
string encryption	0.767	0.694	0.767	0.678	933.559s	100	70
control flow obfuscation	0.796	0.783	0.796	0.779	10.845s	50	90
reflection obfuscation	0.763	0.606	0.763	0.662	211.492s	100	40
average	0.754	0.667	0.754	0.677	463.4s	·	·

Table 4: Experiment results with Extra Trees

	Accuracy	Precision	Recall	F1-score	training time	# of trees	max_depth
identifier renaming	0.701	0.591	0.701	0.578	8.071s	100	90
string encryption	0.770	0.719	0.770	0.671	9.641s	90	40
control flow obfuscation	0.800	0.790	0.800	0.789	0.893s	10	40
reflection obfuscation	0.763	0.582	0.763	0.661	1.227s	50	70
average	0.759	0.671	0.759	0.675	4.958s	·	·

All classifiers underperforms in identifier renaming detection. The detection accuracy ranges 0.688 ˜ 0.701 in identifier renaming detection and 0.722 ˜ 0.800 in the other obfuscation technique detection. This is because identifier renaming techniques rarely modify the structure of program codes. We guess there might be little practical feature in decompiled identifier-obfuscated classes.

Table 6 summarizes the experiment results. It shows the best-performing algorithm for each obfuscation technique detection and its training time. Overall, Extra Trees outperforms other algorithms and has the shortest training times.

5 Conclusion and Future Work

Because malware writers use code obfuscation techniques to evade malware detection, malware analysts need to understand which obfuscation techniques are applied to the malware in order to effectively analyze the obfuscated malware. For analyzing the obfuscated malicious Android apps, previous studies have been conducted to identify obfuscation techniques at an app level by using signature-based pattern matching or machine learning-based classification. These kinds of studies have the limitations that they are ineffective to identify a class-level obfuscation technique as well as a new identification approach is required whenever an obfuscation technique is changed. Some existing machine learning-based methods have needed feature engineering for identifying an obfuscation technique.

In this paper, we have proposed a new framework for identifying obfuscation techniques at a class level not app level in Android. Our framework vectorizes the decompiled codes of each class of Android apps using a paragraph vector. Then the output vectors are fed to four machine classifiers for identifying

Table 5: Experiment results with Linear SVM

	Accuracy	Precision	Recall	F1-score	training time	gamma	C
identifier renaming	0.696	0.484	0.696	0.571	108.563s	0.0001	0.0001
string encryption	0.766	0.587	0.766	0.665	140.059s	0.0001	0.0001
control flow obfuscation	0.722	0.522	0.722	0.606	119.344s	0.0001	0.0001
reflection obfuscation	0.770	0.593	0.770	0.670	11.791s	0.0001	0.0001
average	0.739	0.547	0.739	0.628	94.939	·	·

Table 6: Best-performing algorithm for each obfuscation technique detection

Obfuscation technique	Best-performing algorithm	training time
identifier renaming	Extra Trees	8.071s
string encryption	Extra Trees	9.641s
control flow obfuscation	Extra Trees	0.893s
reflection obfuscation	Random Forest	16.78s

which obfuscation technique is applied to the class. Finally, we analyze the performance of the four classifiers for each obfuscation technique. The proposed framework allows malware analyst to determine whether a malicious app is equipped with an obfuscation technique at a class level and reduce the resources required to analyze the app. Experimental results show that the framework is relatively suitable for identifying control flow obfuscation. Obfuscators can combine several obfuscation techniques and apply the combination to a class. In future, we plan to consider the case where two or more obfuscation techniques are applied to a class.

Acknowledgments

This research was supported by (1) Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science and ICT (no. 2018R1A2B2004830) and (2) the MSIT(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] "AndroGuard," https://github.com/androguard/androguard [Online; accessed on December 15, 2019].
[2] "dashO," https://www.preemptive.com/products/dasho/overview [Online; accessed on December 15, 2019].
[3] "DexGuard," https://www.guardsquare.com/en/products/dexguard [Online; accessed on December 15, 2019].
[4] "DexProtector," https://dexprotector.com/ [Online; accessed on December 15, 2019].
[5] "F-droid," https://f-droid.org/ [Online; accessed on December 15, 2019].
[6] "gensim," https://radimrehurek.com/gensim/ [Online; accessed on December 15, 2019].
[7] "ProGuard," https://www.guardsquare.com/en/products/proguard [Online; accessed on December 15, 2019].
[8] "scikit-learn," https://scikit-learn.org/stable/ [Online; accessed on December 15, 2019].

[9] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, October 2001.

[10] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, September 1995.

[11] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *Proc. of the 14th International Conference on Security and Privacy in Communication Systems (SecureComm'18), Singapore, Singapore*. Springer, December 2018, pp. 172–192.

[12] P. Faruki, H. Fereidooni, V. Laxmi, M. Conti, and M. S. Gaur, "Android code protection via obfuscation techniques: Past, present and future directions," *Computing Research Repository*, November 2016.

[13] Y. Freund and R. Schapire, "A short introduction to boosting," *Journal of Japanese Society for Artificial Intelligence*, vol. 14, no. 5, pp. 771–780, September 1999.

[14] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek, "Obfuscation-resilient, efficient, and accurate detection and family identification of android malware," George Mason University, Tech. Rep., 2015.

[15] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, April 2006.

[16] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. of the 31st International Conference on Machine Learning (ICML'14), Beijing, China*. JMLR W&CP, June 2014, pp. 1188–1196.

[17] O. Mirzaei, J. M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano, "Androdet: An adaptive android obfuscation detector," *Future Generation Computer Systems*, vol. 90, pp. 240–261, January 2019.

[18] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated android malware," in *Proc. of the 7th ACM on Conference on Data and Application Security and Privacy (CODASPY'17), Scottsdale, Arizona, USA*. ACM, March 2017, pp. 309–320.

[19] Y. Wang and A. Rountev, "Who changed you?: Obfuscator identification for android," in *Proc. of the IEEE 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft'17), Buenos Aires, Argentina*. IEEE, May 2017, pp. 154–164.

[20] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in google play," in *Proc. of the 34th Annual Computer Security Applications Conference (ACSAC'18), San Juan, Puerto Rico, USA*. ACM, December 2018, pp. 222–235.

Author Biography

Minjae Park received the B.E degree in the Dept. of Information and Communication Engineering from National Institute for Lifelong Education, Korea in 2015 and his M.E. degree in Computer Science from Dankook University in and 2019. His research interests include computer system security and reverse engineering.

Geunha You is currently an undergraduate student at Dept. of Software Engineering in Dankook University, Korea. His research interests include computer system security, system software, embedded system and reverse engineering.

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 & 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 and his M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 2000 and 2007, respectively. He is now a professor in the Dept. of Software Technology at Konkuk University. His research interests include real-time scheduling and computer security.