An Empirical Evaluation of Software Obfuscation Techniques Applied to Android APKs

Felix C. Freiling, Mykola Protsenko, and Yan Zhuang(✉)

Department of Computer Science,
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany
{felix.freiling,mykola.protsenko,yan.zhuang}@cs.fau.de

Abstract. We investigate the problem of creating complex software obfuscation for mobile applications. We construct complex software obfuscation from sequentially applying simple software obfuscation methods. We define several desirable and undesirable properties of such transformations, including idempotency and monotonicity. We empirically evaluate a set of 7 obfuscation methods on 240 Android Packages (APKs). We show that many obfuscation methods are idempotent or monotonous.

Keywords: Software obfuscation · Mobile security · Android · Software protection · Reverse engineering · Software metrics

1 Introduction

Software obfuscation is a common tool to protect software from reverse engineering, and it is particularly relevant for architectures that execute *bytecode* because bytecode is much easier to decompile (and therefore to reverse engineer) than native machine code. The Android platform is a prominent and practically relevant example of such an environment. In Android, applications (or "apps") are shipped in the Android Package format. An Android Package (APK) contains *dex bytecode* for which it is rather easy to reconstruct the original Java source code using decompilers. There are multiple software obfuscation frameworks for Java in general (such as Sandmark [9]) and Android in particular (such as ProGuard [15]). In this paper, we focus on Pandora [19], an obfuscation framework that contains a representative selection of obfuscating transformations specifically for Android. Pandora is based on the Soot framework [20] which is a Java optimization tool working on source code. Therefore, Pandora has to transform the dex file into a Java archive (jar) file, then applies obfuscation and finally tansforms the resulting program back into a dex file again.

Intuitively, software obfuscation transforms a program in such a way such that

- the original program semantics are preserved (maybe with a negligible delay in performance) and
- the resulting program is harder to understand as the original one.

J. Tian et al. (Eds.): SecureComm 2014, Part II, LNICST 153, pp. 315–328, 2015.
DOI: 10.1007/978-3-319-23802-9_24

The problem of theoretical and practical definitions is to capture what it means for a program to be "harder to understand". While there exist some obfuscation methods that are provably hard to reverse [3,21], the general understanding is that strong obfuscation for general programs is impossible (for reasonable definitions of "strong") [1]. Despite theoretical advances of the field [13,14], we must therefore continue to approximate the strength of practical obfuscation methods *empirically*.

Most practical methods are designed with an "idea" in mind of why the resulting program is harder to understand, but for most techniques there is no empirical evaluation, especially in comparison to other obfuscation methods. Emirically, the hardness to understand a piece of code can only be checked by human experiment [4] or (as an approximation) by using specific software complexity/quality metrics. In this paper, we focus on software obfuscation for the Android platform and empirically evaluate the obfuscation techniques of Pandora [19].

The research question we investigate in this paper is the following: *Considering the basic obfuscation techniques of* Pandora*, does obfuscation improve if we apply the same obfuscation technique multiple times?* Since obfuscation methods are usually applied only once to a piece of code this might appear as a strange and unusual question. However, our aim is not primarily to build better obfuscation techniques but rather to *understand* the behavior of existing techniques better. Rephrasing the question, we ask: What are the characteristics of software if obfuscation methods are reused?

To answer our questions empirically, we have built an Android software obfuscation framework that allows us to automate the task of obfuscation and software complexity measurement. In designing this framework, we formalized the problem of building complex obfuscation methods from simpler ones. This allowed us to identify a set of desirable properties which practical program transformations should satisfy and to classify the investigated obfuscation techniques in this respect.

In this paper we make the following contributions:

1. We formally define what it means to apply a sequence of obfuscation methods to a program and identify desirable and undesirable structural properties. For example, we identify *idempotency* and *monotony* as desirable properties of obfuscation functions.
2. We empirically evaluate 7 obfuscation methods with respect to 8 software complexity metrics on a set of 240 Android Packages (APKs). Following our research question, we restrict our investigation to properties inherent to a single obfuscation function, i.e., we only investigate iterative applications of the same obfuscation methods to a given program.
3. We show that most obfuscation methods exhibit "stable" properties when used iteratively, i.e., they are idempotent or monotonous. However, a single obfuscation method usually exhibits different stable properties with respect to different complexity metric, i.e., it might be idempotent regarding one metric and monotonous regarding a second metric.

Related Work. According to Collberg and Nagra [7], to measure the strength of practical obfuscation techniques, a definition is required that allows to compare the *potency* of two transformations. Preda and Giacobazzi [10], for example, give a definition that classifies a transformation as potent when there exists a property that is not preserved by the transformation. Of course, *some* properties of a program must be preserved since the obfuscated program should compute the same functionality. In practice, however, the definition of Preda and Giacobazzi [10] only allows to compare simple transformations in isolated environments. To compare the strength of two obfuscated real-world programs, their framework cannot be applied.

Seminal work of Collberg et al. [8] surveyed different obfuscation techniques and classified them mainly into three categories: data obfuscations, control obfuscations, and layout obfuscations. They also investigated the effect of single obfuscation steps on different software metrics and even proposed a Java obfuscation framework (named Kava [8, Sect. 3]) designed to systematically obfuscate a program such that certain quality criteria are satisfied. We are, however, not aware of any empirical evaluation of the framework.

Outline. This paper is structured as follows: We first give some background in Sect. 2. We then define desirable properties of obfuscation functions in Sect. 3. After giving an overview of our obfuscation framework in Sect. 4 we provide the results and a discussion in Sect. 5. Section 6 concludes the paper.

2 A Brief Tour of the Obfuscation and Metric Zoo

This section provides a brief overview of the obfuscation techniques and the software complexity metrics we used in our evaluation. Where appropriate, we give an akronym as a shorthand in the later discussion.

Obfuscation Methods. In total, we considered 10 obfuscation techniques that were available in the Pandora obfuscation tool [19] for Android. These techniques represent a broad selection of specialized and general obfuscation methods. They can be classified into 4 transformations at the method level and 6 transformation at the class level.

At the method level, we considered the following techniques:

- *Array index shift* obfuscates the use of the arrays by shifting the indices with a constant shift value.
- *Compose locals* groups the method's local variables of the same type and composes them to a single container variable, such as array or map. For the latter one random keys of the types string, character or integer are used.

At the class level, we considered the following obfuscation methods:

- *Drop modifiers* is one of the most simple transformations: It discards the access-restricting modifiers like `private`, `protected`, or `final` for classes, methods, and fields.

- *Extract methods* "outlines" the bodies of the methods, the signatures of which cannot be changed due to restrictions laid down by the application design, e.g., methods called in response to the system and user events, like the `onCreate` of the Android `Activity` class.
- *Move fields* changes the hosting class of the field and replaces all accesses correspondingly. Note that for non-static fields a reference object to the new host class is required. Such objects are created and initialized in each constructor of the class.
- In analogy to move field, *move methods* moves a method from one class to another. If the method makes use of the implicit `this` parameter, it must be added to the explicit parameter list of the method.
- *Merge methods* replaces two methods of the same class and the same return type with a single method. It interleaves both parameter lists and the bodies of the methods. Furthermore, an additional `key` parameter is added in order to differentiate which of the bodies is to be executed at runtime.

Software Complexity Metrics. The set of software complexity metrics used in our evaluation contains measurements of the control flow complexity and data flow complexity of the methods, as well as the usual object-oriented design (OOD) metrics suite. We now give a very brief description of these metrics.

At the method level, we used two metrics to measure the complexity of the control flow and data flow: McCabe's Cyclomatic Complexity and Dependency Degree, respectively. Cyclomatic Complexity [16] of the method corresponds to the number of the linearly independent circuits in the control flow graph (CFG). It can be computed as $v = e - n + p$, with e, n, and p being the number of edges, nodes, and connected components of the CFG respectively.

The Dependency Degree metric (abbreviated as DepDegree), proposed by Bayer and Fararooy [2], is defined with help of the *dependency graph*, which is constructed for the given CFG as follows. The nodes correspond to the instructions of the method, and the edges reflect the dependencies between them. One instruction is said to depend on the another if it uses some values defined by that instruction. Then, the Dependency Degree of the method is defined as the number of edges in the corresponding dependency graph.

The measurement of OOD complexity is performed with a suite containing 6 metrics by Chidamber and Kemerer [5]. These metrics measure complexity on the class level:

- *Weighted Methods pro Class* (WMC): the number of the methods defined in a class.
- *Depth of Inheritance Tree* (DIT): the number of classes along the path from the given class to the root in the inheritance tree.
- *Number of Children* (NOC): the number of direct subclasses.
- *Coupling Between the Object classes* (CBO): the number of coupled classes. Here, two classes are considered coupled, if one of them uses methods or instance variables of the other one.
- *Response Set for a Class* (RFC): the number of declared methods plus the number of methods the declared ones call.

– *Lack of Cohesion in Methods* (LCOM): Given a class C declaring methods $M_1, M_2, ..., M_n$ and a set of instance variables used by the method M_j denoted as I_j, define $P = \{(I_i, I_j) | I_i \bigcap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) | I_i \bigcap I_j \neq \emptyset\}$. Then $LCOM = max\{|P| - |Q|, 0\}$.

3 Obfuscation as a Function

In this section, we formally define what it means to apply a sequence of obfuscation methods to a program and identify desirable and undesirable structural properties.

(Complex) Obfuscation Methods. We consider a finite set of obfuscation methods $\Omega = \{\omega_1, \omega_2, \ldots\}$. Each such method is defined as a program transformation for a particular domain. Let $\mathcal{P}$ denote the set of all programs from that domain (e.g., all programs written in C). Then formally, every ω_i is defined as a function

$$\omega_i : \mathcal{P} \mapsto \mathcal{P}$$

such that $\omega_i(p)$ computes the same functionality as p without being exponentially slower or larger than p.

A complex obfuscation method can be defined by applying first a specific simple obfuscation method ω_i and then another simple obfuscation method ω_j. From the perspective of obfuscation as a function this is the *composition* of functions, i.e., $\omega_j(\omega_i(p))$.

We now define what it means to apply a *sequence* of obfuscation methods from Ω to a program. We denote by Ω^+ the set of all finite sequences of elements of Ω (including the empty sequence). An example sequence is $\langle \omega_1, \omega_2, \omega_1 \rangle$. For two sequences $\alpha, \beta \in \Omega^+$ we denote by $\alpha \cdot \beta$ the concatenation of α and β and by $\alpha \sqsubset \beta$ that α is a strict (i.e., shorter) prefix of β.

We now define a general notion of obfuscator composition $\mathcal{O}$ that uses individual methods from Ω to create new (and possibly better) variants of obfuscation methods.

Definition 1. Complex obfuscator composition *is a function* $\mathcal{O} : \mathcal{P} \times \Omega^+ \mapsto \mathcal{P}$ *that satisfies the following conditions for all* $p \in \mathcal{P}$*:*

1. $\mathcal{O}(p, \langle\rangle) = p$
2. *For all* $\alpha \in \Omega^+$*:* $\mathcal{O}(p, \alpha \cdot \omega) = \mathcal{O}(\omega(p), \alpha)$

As an example, let $\alpha = \langle \omega_1, \omega_2, \omega_1, \omega_3 \rangle$. Then we have:

$$\mathcal{O}(p, \alpha) = \omega_1(\omega_2(\omega_1(\omega_3(p))))$$

Properties. We now define a set of properties that complex obfuscation methods can satisfy. These properties sometimes refer to software metrics such as those defined in Sect. 2. We formalize them as a finite set of functions $M = \{m_1, m_2, \ldots\}$. Each metric is a function that takes a program and maps

to into a totally ordered metrical space like the natural numbers. A simple complexity metric is *lines of code* for which the metrical space is the set of natural numbers and the ordering relation is $\leq$.

The first property of idempotency refers to the effect of an individual obfuscation function and intuitively states that applying the function more than once does not improve the obfuscation result regarding a particular metric.

Definition 2. *An obfuscation function ω is* idempotent with respect to metric $m \in M$ *iff for all $p \in \mathcal{P}$ holds that $m(\omega(\omega(p))) = m(\omega(p))$.*

Note that some obfuscation methods such as *drop modifiers* are idempotent for all metrics since they satisfy the stronger property of $\omega(\omega(p)) = \omega(p)$. We call such methods simply *idempotent*. An obviously non-idempotent method is *extract methods*, since it creates a new methods with every invocation.

We now define an additional property of obfuscation functions: monotony. Intuitively, monotonous obfuscation functions continuously increase (or decrease) the value of the considered metrics. Obviously, all idempotent obfuscators are monotonous. For non-monotonous obfuscators, there are metrics which are unstable, i.e., which rise and later decrease or vice versa.

Definition 3. *An obfuscation function ω* is monotonous with respect to metric $m \in M$ *iff the following holds: Let α and β be sequences of ω: If $\alpha \sqsubset \beta$ then $m(\mathcal{O}(p, \alpha)) \leq m(\mathcal{O}(p, \beta))$, where $\leq$ is the order on the metrical space of m.*

While idempotency is rather easy to understand, monotony is more complex since the effect of an obfuscation method on a particular metric is not always clear. In general, we believe that "good" obfuscation methods should be either idempotent or at least monotonous for most metrics considered. Idempotency is a good property because it makes an obfuscation method easy to apply and it facilitates control of its effects. Monotony is positive, because it corresponds to the expectation that obfuscation methods make analysis (increasingly) "harder".

We will evaluate the obfuscation methods presented in Sect. 2 and show that most of them satisfy these desirable properties.

4 An Android Obfuscation Framework

We have designed and implemented a framework with which we can automatically apply and evaluate complex obfuscation methods composition to Android APKs. The structure of the framework is depicted in Fig. 1. The framework is based on three modules which we use to apply and evaluate obfuscation methods:[1]

1. Pandora [19] is an Android obfuscation tool that implements the obfuscation techniques introduced in Sect. 2. It operates on jar files that are the main ingredient of APKs.

[1] Although we rely heavily on Pandora, please note that our framework does not implement any obfuscation methods or metrics itself and can be extended with other obfuscation tools (such as Sandmark) later. It therefore is rather a "meta framework".

2. SSM, a supplementary function to PANDORA [19], is a measurement tool for computing the different software complexity metrics mentioned in Sect. 2. SSM computes these metrics on every class and method in a jar file.
3. Androsim [18] is a tool written in Python for measuring the similarity between APKs. It is part of the Androguard APK analysis toolset [11]. The basis for similarity measurement is the *compression distance* between different APKs. To compute this distance, APKs are compiled into an intermediate representation focusing much on the control flow and abstracting from identifier names. The resulting strings of the two APKs are then compared by computing the *normalized compression distance* (NCD) [6]. We use the resulting value as a reference value for the SSM metrics.

We now explain how APKs are processed in our framework. The starting point is a set of APKs stored in the file system of our analysis machine (top of Fig. 1). The APK is transformed into a jar file that is subsequently processed by PANDORA. The resulting jar file, which is a obfuscated version of the original jar file, goes through the post-processing section (e.g., it is checked whether it contains valid JVM code using the *jasmin* tool [17]). To distinguish both files in the file system, we append the name of the applied obfuscation method to the original filename of the jar file ($*\_\omega.jar$, where ω is an identifier of the obfuscation method record in our database). After that, the jar file is turned into a dex file and compressed into an Android package which is again stored in the file system. In the meantime, we use Androsim and SSM to compare and compute different metrics on the original and transformed jar files and APKs (see Fig. 1). All the generated data is inserted into a database.

The above processing refers to one step of the complex obfuscation composition of Definition 1. To apply multiple obfuscation techniques in sequence, the framework will perform the same iteration with another obfuscation techniques to the same APK. The selection of the input files as well as the iteration and

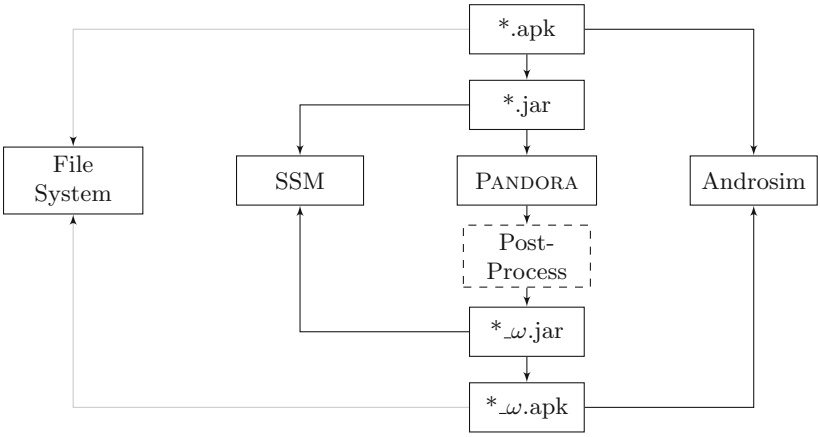

Fig. 1. Structure of the Android Obfuscation Framework.

type of obfuscation techniques applied on the APKs are totally automated using Python.

5 Results

We applied the obfuscation transformations described in Sect. 2 to 240 APKs which we randomly selected from a set of more than 1000 APKs which we downloaded from the open source Android application market *F-Droid* [12]. With more time we would have chosen more APKs for the computation of our results but we consider 240 a large enough set such that our results have some significance.

Idempotent Transformations. As introductory example for a clearly idempotent transformation, we show two metrics (CBO and LCOM) of *drop modifiers* in

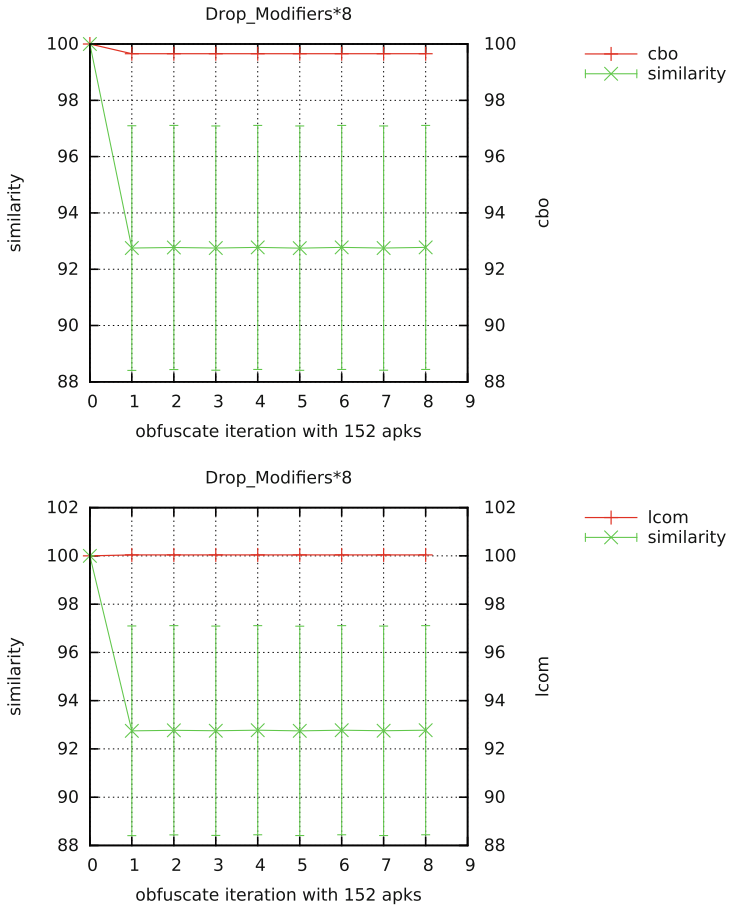

Fig. 2. CBO and LCOM measurements of *drop modifiers* (Color figure online).

Fig. 2. In these (and the following) graphs, the horizontal axis denotes the number of transformation iterations and the vertical axis denotes the percentage of the original APK's complexity or similarity. The red and green lines correspond to the given complexity metric and the similarity measured with Androsim,

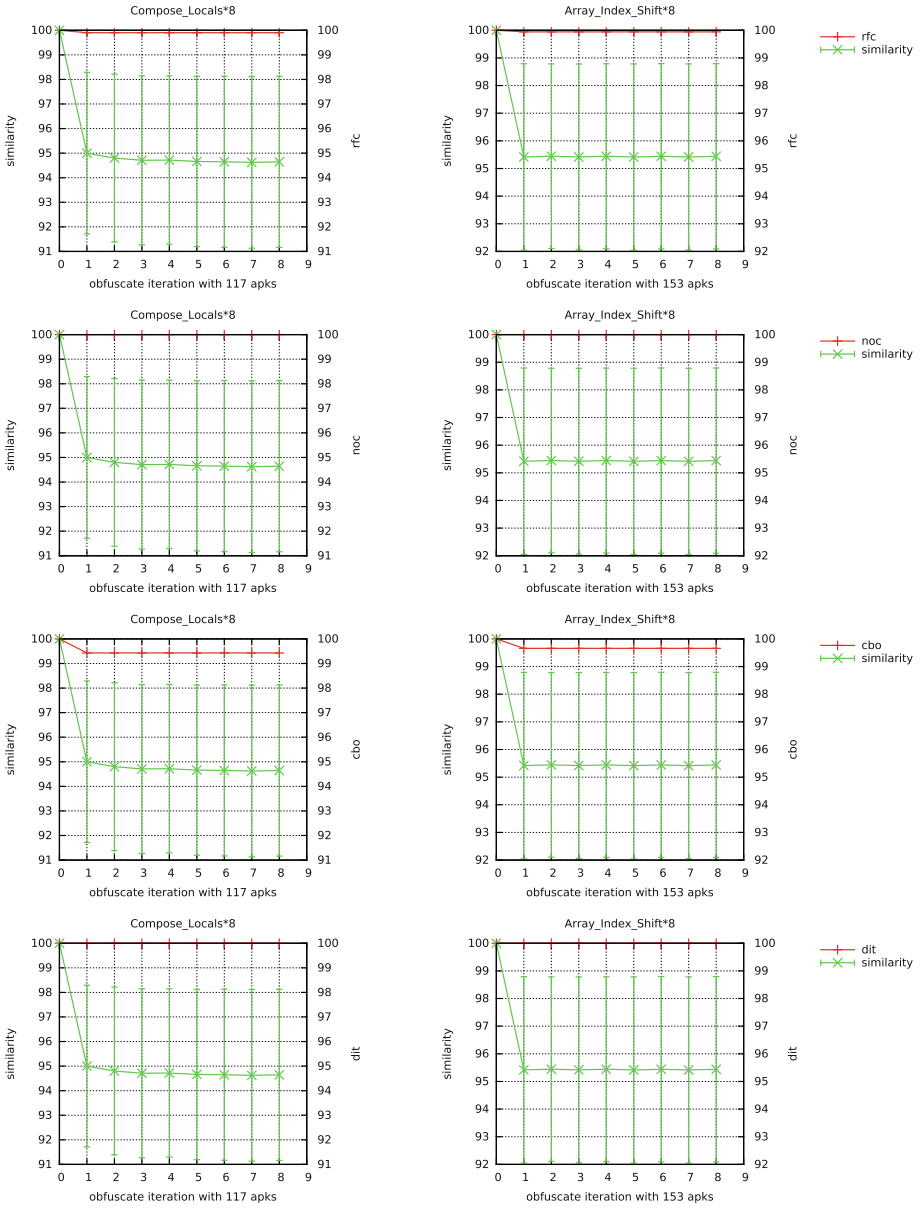

Fig. 3. Selection of OOD metrics for *compose locals* (left) and *array index shift* (right).

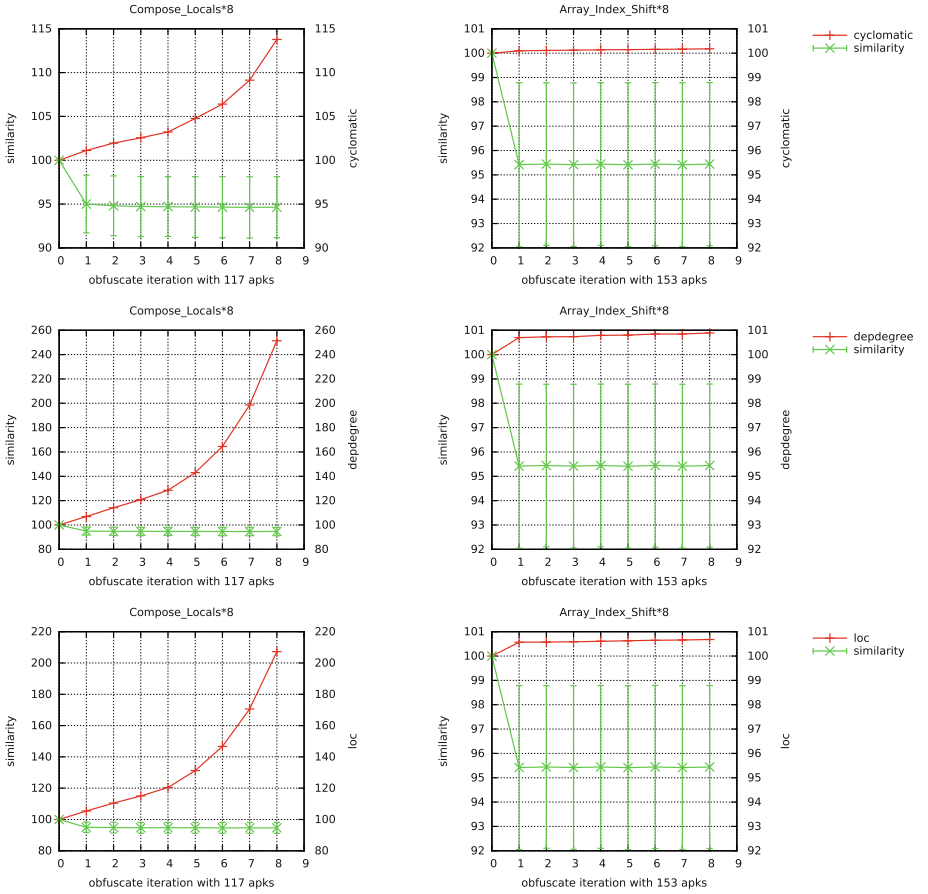

Fig. 4. Cyclomatic complexity, DepDegree and LOC metrics for *compose locals* (left) and *array index shift* (right).

respectively. The values of similarity serve the reference purposes and indicate the overall change of the program structure caused by obfuscation.[2]

Transformations applied on the intraprocedural level, namely *compose locals* and *array index shift*, are as expected idempotent with respect to all OOD metrics. *Move fields* is idempotent for all metrics. *Move methods* exhibits idempotency for cyclomatic complexity and WMC, since it neither changes the code of the methods nor their overall number.

Transformations applied on the intraprocedural level, namely *compose locals* and *array index shift*, are as expected idempotent with respect to all OOD metrics. This can be seen in Fig. 3.

[2] In the following figures, we have scaled down the graphs to improve the visual "overview" impression with multiple graphs on one page. The caption repeats the method and metric for readability.

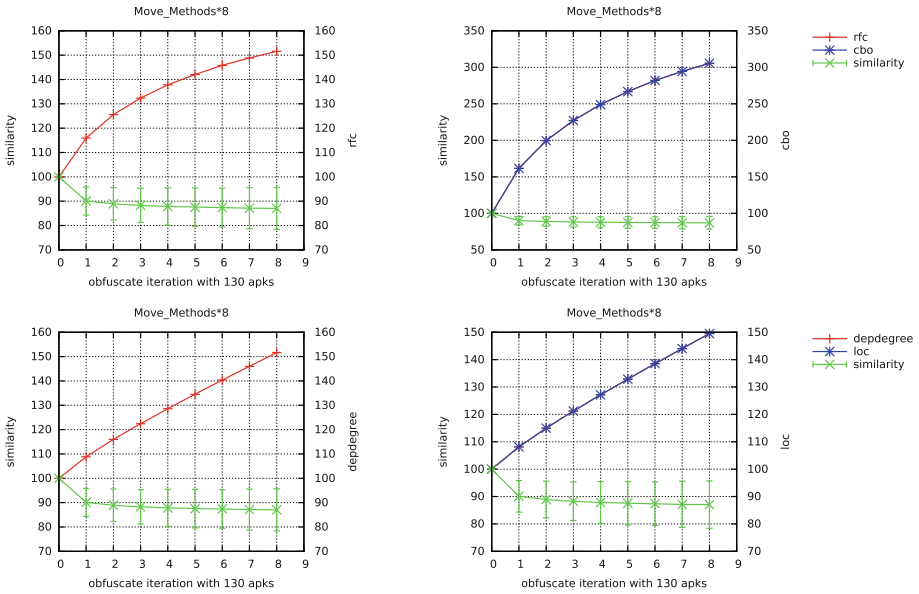

Fig. 5. RFC, CBO, DepDegree and LOC metrics for *move methods*.

Monotonous Transformations. Many of the evaluated obfuscation transformations were found to be monotonous, i.e., their continuous application keeps increasing some of the complexity metrics. In particular, the method-level obfuscations *compose locals* and *array index shift* are monotonous for the method-level metrics Cyclomatic complexity, DepDegree and LOC (shown in Fig. 4). For *compose locals* the complexity growth was superlinear for all three metrics, whereas *array index shift* shows very slow linear increase reaching less than 101 % of the original complexity after 8 transformations.

Move methods is monotonous for the OOD metrics RFC and CBO as well as DepDegree and LOC (in Fig. 5). As mentioned earlier, with respect to the cyclomatic complexity this transformation is idempotent. This is because moving non-static methods requires a reference object to the target class, which is stored in the class field and copied to the local variable before the method invocation. Since this does not add any branches, cyclomatic complexity stays unchanged, however, new instructions and variables are added, which increases the other two method-level metrics.

Merge methods is monotonous with respect to DepDegree and LOC: It adds new instructions and operations on variables. However, allthough it adds an additional branch per merge operation, the cyclomatic complexity (shown in Fig. 6) remains constant, since the number of circuits in the merged code equals the sum of the circuits of the merged methods.

Unstable Transformations. Some of the obfuscation transformations did not fit the definitions of monotonicity or idempotency for certain metrics. These

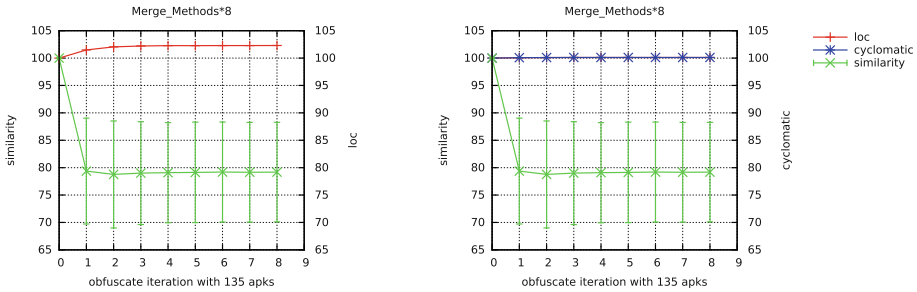

Fig. 6. *Merge methods*: measurements for metrics LOC (left) and Cyclomatic complexity (right).

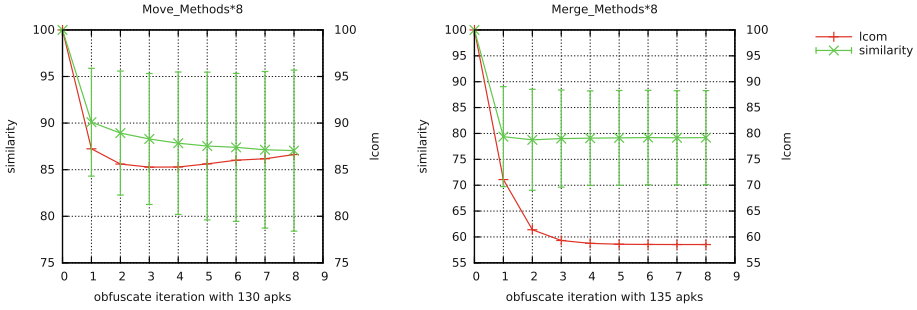

Fig. 7. A comparison of LCOM for *move methods* (left) and *merge methods* (right).

unstable transformations are particularly interesting. One example is *merge methods* which exhibits monotonicity and idempotency for CBO and DepDegree, respectively, but is unstable in WMC, RFC, and LCOM (see Fig. 7, right). *Move methods* showed interesting unstable behavior with respect to the LCOM too: After decreasing within the first 3 obfuscation runs, LCOM increases again (see Fig. 7, left). The key to understanding of this phenomenon lies in the random application of the transfomation and the nature of the metric: Recall that LCOM increases with the number of class methods operating on different sets of instance variables and decreases with the number of those operating on the intersecting sets of instance variables. Since the movement process of *move methods* is randomized, a repeated application of the transfomation can result in both more and less cohesive method layouts, therefore decreasing or increasing the metric.

6 Conclusions and Future Work

In this paper, we experimentally evaluated obfuscation methods when they are applied iteratively and we defined and revealed some structural properties of these methods regarding different software complexity metrics. While the results

are interesting and show that most obfuscation methods we have used exhibit rather "stable" properties, the general picture is rather complex since a single obfuscation method usually exhibits different properties (i.e., monotonicity or idempotency) regarding different complexity metrics. Interestingly, a few obfuscation methods have unstable properties regarding some of the metrics.

More experiments are needed to understand this behavior better. A more thorough understanding of the behavior of obfuscation methods is the basis for a more intelligent application of these methods in practice. For example, following an idea of Collberg et al. [8], a detailed understanding of the effects of certain obfuscation methods on complexity metrics would allow to transform programs in such a way that specific "target" complexity requirements are reached with a minimum number of obfuscation steps. Our obfuscation framework is a good basis for such investigations.

Acknowledgments. We wish to thank Tilo Müller for his comments on a prior version of this paper. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89), the "Bavarian State Ministry of Education, Science and the Arts" as part of the FORSEC research association, and by a scholarship of the Chinese State Scholarship Fund.

References

1. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (im)possibility of obfuscating programs. J. ACM **59**(2), 6 (2012)
2. Beyer, D., Fararooy, A.: A simple and effective measure for complex low-level dependencies. In: Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, ICPC 2010, pp. 80–83. IEEE Computer Society, Washington, DC (2010)
3. Canetti, R., Dakdouk, R.R.: Obfuscating point functions with multibit output. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 489–508. Springer, Heidelberg (2008)
4. Ceccato, M., Penta, M., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empirical Softw. Eng. **19**(4), 1040–1074 (2013)
5. Chidamber, S.R., Kemerer, C.F.: Towards a metrics suite for object oriented design. SIGPLAN Not. **26**(11), 197–211 (1991)
6. Cilibrasi, R., Vitányi, P.M.B.: Clustering by compression. IEEE Trans. Inf. Theory **51**(4), 1523–1545 (2005)
7. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking and Tamperproofing for Software Protection. Addison-Wesley Longman, Amsterdam (2009)
8. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, The University of Auckland, July 1997
9. Collberg, C.S., Myles, G., Huntwork, A.: Sandmark-a tool for software protection research. IEEE Secur. Priv. **1**(4), 40–49 (2003)

10. Dalla Preda, M., Giacobazzi, R.: Semantics-based code obfuscation by abstract interpretation. J. Comput. Secur. (JCS) **17**(6), 855–908 (2009)
11. Desnos, A., Gueguen, G.: Android: From reversing to decompilation. In: Black Hat, Abu Dhabi (2011)
12. F-Droid Ltd., F-droid. https://f-droid.org/
13. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: FOCS, pp. 40–49. IEEE Computer Society (2013)
14. Gentry, C.: Fully homomorphic encryption using ideal lattices. In Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing (STOC), pp. 169–178. ACM, New York (2009)
15. Lafortune, E.: Proguard homepage, June 2014. http://proguard.sourceforge.net/
16. McCabe, T.: A complexity measure. IEEE Trans. Softw. Eng. **SE-2**(4), 308–320 (1976)
17. Meyer, J., Reynaud, D.: Jasmin. http://jasmin.sourceforge.net/
18. Pouik and G0rfi3ld: Similarities for fun and profit, April 2014. http://phrack.org/issues/68/15.html
19. Protsenko, M., Müller, T.: Pandora applies non-deterministic obfuscation randomly to android. In: MALWARE, pp. 59–67. IEEE (2013)
20. Vallée-Rai, R., Gagnon, E., Hendren, L.J., Lam, P., Pominville, P., Sundaresan, V.: Optimizing java bytecode using the soot framework: is it feasible? In: Watt, D.A. (ed.) CC/ETAPS 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
21. Wee, H.: On obfuscating point functions. In: Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing (STOC), Baltimore, MD, USA, pp. 523–532. ACM, April 2005