A Framework for Evaluating Mobile App Repackaging Detection Algorithms

Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu

The Pennsylvania State University,
{hhuang,szhu}@cse.psu.edu, {pliu,dwu}@ist.psu.edu

Abstract. Because it is not hard to reverse engineer the Dalvik bytecode used in the Dalvik virtual machine, Android application repackaging has become a serious problem. With repackaging, a plagiarist can simply steal others' code violating the intellectual property of the developers. More seriously, after repackaging, popular apps can become the carriers of malware, adware or spy-ware for wide spreading. To maintain a healthy app market, several detection algorithms have been proposed recently, which can catch some types of repackaged apps in various markets efficiently. However, they are generally lack of valid analysis on their effectiveness. After analyzing these approaches, we find simple obfuscation techniques can potentially cause false negatives, because they change the main characteristics or features of the apps that are used for similarity detections. In practice, more sophisticated obfuscation techniques can be adopted (or have already been performed) in the context of mobile apps. We envision this obfuscation based repackaging will become a phenomenon due to the arms race between repackaging and its detection. To this end, we propose a framework to evaluate the obfuscation resilience of repackaging detection algorithms comprehensively. Our evaluation framework is able to perform a set of obfuscation algorithms in various forms on the Dalvik bytecode. Our results provide insights to help gauge both $broadness$ and $depth$ of algorithms' obfuscation resilience. We applied our framework to conduct a comprehensive case study on AndroGuard, an Android repackaging detector proposed in Black-hat 2011. Our experimental results have demonstrated the effectiveness and stability of our framework.

Keywords: Mobile apps, reverse engineering, repackaging, obfuscation resilience, malware

1 Introduction

In the past years, mobile phone sales have grown extremely fast and Android [8] has become the dominant of the mobile device market. This gives a burst of new applications pushed into the Android Market. In the end of 2012, the number of apps reached 700,000; however, Google provides little vetting on these apps to prevent it from plagiarism or malicious repackaging.

There are mainly two motivations for app repackaging. First, dishonest developers may repackage others' apps under their own names or embed different advertisements, and then republish them to the app market to earn monetary profit. Second, malware

writers modify a popular app by inserting some malicious payload into the original program. The purpose is to take over mobile devices, steal users' private information, send premium SMS text messages stealthily, or purchase apps without users' awareness. They leverage the popularity of the original program to increase the propagation of the malicious one. Both types of repackaging are severe threats to the app markets. Even without consideration of code obfuscation, it has been found that about 5% to 13% of apps in third party app markets are the plagiarism of applications in the official Android market [29]. Besides, according to a recent study [30], among the analyzed 1260 malware samples, the authors found that 1083 of them (or 86.0%) were repackaged versions of legitimate apps with malicious payloads, indicating repackaging is a favorable vehicle for mobile malware propagation. However, as the commercial motivation grows, nothing prevents plagiarizers and repackagers using code obfuscation techniques to evade detection. Moreover, since users can download applications from both official market Google Play and third party markets in different countries ($e.g.$, Anzhi, a big Chinese Android app market), the repackaging problem can appear both inter- and intra- market, which increases the scale and challenge for repackaging detection.

Due to the very large number of applications in the market and easiness for reverse engineering and manipulation of Dalvik bytecode, several researchers have proposed detection schemes based on static analysis on DEX file [29], [19], [23]. Static code analysis based detection is more efficient than the dynamic ones. However, in practice, sophisticated code obfuscations can be easily applied to evade static analysis based detections [28], and such obfuscation techniques can be easily adapted to mobile applications scenario. When applied to mobile applications, they can greatly increase false-negative rates of the existing detection algorithms. Moreover, in these works, manual inspections are often used to check the false positives in their results. In general, all the detection algorithms currently pay more attention to the computational efficiency of their algorithms and are lack of a comprehensive analysis on algorithm accuracy. Hence, they can be very vulnerable against obfuscation based repackaging.

In this work, we propose a framework to automate the evaluation of repackaging detection algorithms against various obfuscation techniques. Our paper makes the following contributions:

1. We perform a survey study on the existing major repackaging detection algorithms, their evaluation methods, and provide insights on their pros and cons;
2. We take the first step in this field to provide an evaluation framework to measure the obfuscation resilience of detection algorithms;
3. We design our framework by gluing seamlessly all the bytecode conversion and obfuscation tools together. The effectiveness and stability of current framework are fully tested and evaluated;
4. To measure the obfuscation resilience of detection algorithms in a comprehensive way, in our framework, we propose the notion of broadness and depth analysis. We perform a case study with our tool on an open source detection algorithm AndroGuard [20] from Blackhat 2011. Our evaluation results show that while AndroGuard demonstrates reasonable strength against many obfuscation techniques, it is very vulnerable to obfuscation relevant to control flow manipulation performed

on the method granularity, and multiple obfuscations when combined can further decrease its detection capability.

The remainder of the paper is structured as follows. Section 2 provides a study on a number of obfuscation detection algorithms. According to our observation from the study, an evaluation framework on the algorithms' obfuscation resilience has been proposed in Section 3. Section 4 describes the current setup of our framework. Then our experimental result of using the framework to conduct one case study on AndroGuard has been presented. We review related works in Section 5 and conclude with Section 6.

2 Study of existing repackaging detection algorithms

In this section, we first explain with a toy example how to conduct the Dalvik bytecode manipulation. Then we perform a study on several recently proposed algorithms for Android application repackaging detection, including Fuzzy Hashing based detection [29], Program Dependence Graph (PDG) based detection [19] and Feature Hashing based detection [23].

2.1 Background on Dalvik bytecode

----------The original bytecode pattern from Skype classes.dex ----------

1. *invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;*
2. *move-result-object v1*
3. *const-string v2, "TYPE"*
4. *invoke-interface {v0, v1, v2}, Ljava/util/Map;->*
   put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;

--------The semantics-preserving bytecode pattern after manipulation--------

1. *invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;*
2. *move-result-object v1*
3. *move-object v3, v1* <<----- use extra virtual register "v3"
4. *const-string v2, "TYPE"*
5. *move v4, v2* <<----- use extra virtual register "v4"
6. *invoke-interface {v0, v3, v4}, Ljava/util/Map;->*
   put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
7. *move v2, v4* <<----- update the register "v2" according to register "v4"
8. *move-object v1, v3* <<----- update the register "v1" according to register "v3"

Fig. 1. An example of Dalvik bytecode manipulation

An Android application package is called *.apk* file, which must contain the program's Dalvik bytecode, resources and a XML manifest file. Each Android application is initially developed as a Java program. It is compiled into Dalvik bytecode, and then packaged into the $classes.dex$ file as a Dalvik EXecutable (DEX file) to be executed in

Dalvik Virtual Machine (DVM) [2]. The whole compiling process includes two phases: (1) Java code compiled into an intermediary representation (IR), the Java bytecode format, which produces a set of $.class$ files; (2) the IR code is further compiled into Dalvik bytecode by a utility called *"dx tool"*, which produces $classes.dex$, the DEX file.

During the reverse engineering process, a plagiarist first unpacks the *.apk* file of the downloaded Android application and extracts the $classes.dex$ file that includes the actual bytecode for execution. $Classes.dex$ can be dissembled by Baksmali [12] into IR format. After manipulation and obfuscation, it is finally assembled back into an updated $classes.dex$ by Smali. Dalvik bytecode contains more semantic information and provides a higher programming abstraction for the developer than machine code instructions, ($e.g.$, x86 assembly code). Therefore, it is easy for human analysis, reverse engineering and manipulation.

In Figure 1, we show an example on semantic preserving manipulation of Dalvik bytecode. The dissembled bytecode snippet is from the Skype app for Android platform, and it is a representative function invocation. Similar code patterns can be identified all over the program. We manipulate it by using extra virtual parameter registers $v3$ and $v4$. The corresponding output bytecode is shown in the lower part of Figure 1. This code manipulation is semantic preserving, as the value in registers $v1$ and $v2$ are moved into the two extra registers and restored after the execution of opcode *[invoke-interface]*. The *[invoke-interface]* is executed using the extra virtual registers, instead of $v1$ and $v2$ originally. All these manipulations have no side-effect on the current and following context of the program that might use $v1$ and $v2$. The dissembled Dalvik bytecode contains lots of similar function invocation code patterns, so this type of noise instructions can be inserted with a very high frequency throughout the program.

2.2 Fuzzy hashing based detection

In order to measure the similarity between plaintiff and repackaged applications, DroidMOSS [29] leverages specialized hashing technique, called fuzzy hashing. Instead of computing a hash over the entire program instruction set, a hashing value is computed for each local unit of opcode sequence of the $classes.dex$. It uses a reset point to split long opcode sequences into small units and then concatenate all the hash values into a whole. In this way, it can localize the modification caused by repackaging. Also, DroidMOSS focuses on instructions' opcode part in order to be resilient against "oprand string literal" based obfuscation. DroidMOSS can efficiently identify those pieces that were not touched by the repackager and works well when code manipulation was only performed at a few interesting points, *e.g.*, hard coded URLs. For this particular type of repackaging, DroidMOSS has a very high true positive rate.

Among all the detection algorithms we studied, only DroidMOSS has a measurement on its false negative rate through experiments. Two major reasons were reported to lead to potential false negatives. One is that some repackaging cases insert a large chunk of code as noise into the original app and the other fact is that the incomplete white-list of the ad libraries, which produces lots of noise into the opcode sequence. All the noise can result in considerable difference in the final fingerprints.

In Figure 1, we demonstrate that adding extra semantic preserving noise opcodes is not hard. For instance, if one performs the similar code manipulation frequently, the

concatenated hashing value in DroidMOSS can be changed dramatically. Since local hashing value of code snippet unit *[invoke-static → move-result-object → const-string → invoke-interface]* and the corresponding manipulated opcode unit *[invoke-static → move-result-object → move-object → const-string → move → invoke-interface → move → move-object]* are very different, by concatenating all the different pieces into a final hash result, the detection can be evaded. The reset points DroidMOSS they uses to split the whole opcode sequence into small opcode units are semantically irrelevant, that is not depending on basic blocks, or other semantic information of the program. Therefore, the detection can be further evaded by carefully crafting the code manipulation pattern and make the inserted opcode hit the predefined reset points. In this way, the overall opcode structure of the fuzzy hashing computation is much modified, but the semantic of the Dalvik bytecode is still preserved.

2.3 PDG based detection

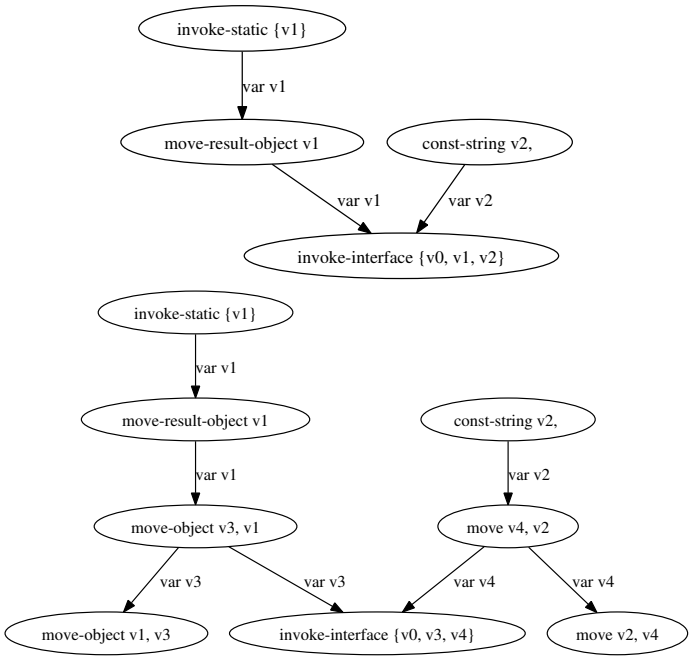

Fig. 2. The example of Data Dependency Graph (Top : original; Below : manipulated)

In DNADroid [19], the *dex* file of Android application is converted to Jar through a tool called *dex2jar*, so that they can leverage WALA [14] to compute the static data dependency graph (DDG) of every method. DDG is considered as the main characteristic of the apps for similarity comparison. DNADroid compares the DDGs within a pre-computed cluster of Android apps using graph isomorphism based algorithms.

Specifically, each vertex in a DDG is a bytecode instruction and each edge initiate from one source instruction to one destination instruction. For example in Figure 2, the instruction *[move-object v3, v1]* is considered as the source instruction for both destination instructions *[invoke-interface v0, v3, v4]* and *[move-object v3, v]*. Since the source assignment instruction ($v3 := v1$) has a side effect on a variable $v3$, and also that $v3$ is used later in the following destination instructions, based on their algorithm, we should link the source instruction to both destination instructions by outgoing edges.

In general, static DDG is resilient against several control flow obfuscations and noisy code insertion attacks that do not modify the data dependency. However, the false positive rate is not evaluated. Indeed, some specific data dependency obfuscations can be designed to evade this approach. Figure 1 shows the data dependency graphs of the toy example in Figure 2 before and after code obfuscation. By comparison, we can observe a dramatic change of data dependency relationship between instructions. This side-effect free manipulation has the potential to evade the graph isomorphism algorithm based detection.

2.4 Feature hashing based detection

Juxtapp [23] is a code-reuse detection scheme based on feature hashing. Similar to DNADroid, the unlabeled $classes.dex$ files of apps are grouped based upon some predefined criteria, to reduce the comparison overhead. *k*-grams of various opcode sequence patterns within each basic block of the program are considered as features. For example, they choose *5*-gram as a moving window of size *5*, which moves within each basic block to map and flag the features into a *m* bit vector. Then the bit vectors are further combined into a feature metric to fingerprint each app. Juxtapp currently uses various predefined opcode sequences as features. For instance, when *[new-instance* $\rightarrow$ *const-string projectSpinnerPos* $\rightarrow$ *invoke-direct* $\rightarrow$ *iget-object* $\rightarrow$ *invoke-static]* appeared in a particular basic block sequentially (with a window size of five), the corresponding feature bit in the bit-vector indicating this opcode sequence feature is flagged with *one*. This detection scheme is able to effectively detect various code reuse cases, including piracy and code repackaging, malware existence, vulnerable code; however, this work does not perform evaluation on the tool's false negative rate.

By using the code manipulation shown in Figure 1, it can potentially destruct the normal opcode pattern of Dalvik bytecode in a very dense fashion. The special features of the program can be normalized by inserted instructions, creating lots of fake feature bits. Both can lead to a high false negative rate of their detection algorithm. Note that although Juxtapp can reduce the noise-injection caused false negatives by decreasing the size of its sliding window for feature definition, this on the other hand reduce the whole feature space and lead to more false positives. To the extreme, Juxtapp can choose window size of one and use *[new-instance]* as one of the features. Then every basic block unit from different apps will probably have this opcode appeared at least once. Thus lots of similar feature metric can be produced for independently developed apps.

In general, we consider it will be very beneficial to tune Juxtapp against various obfuscation techniques and find an optimal way to define the size of its sliding window and the unique feature set.

3 Evaluation framework

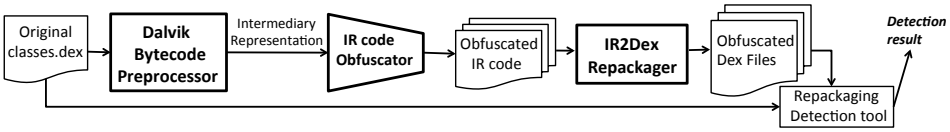

Fig. 3. A Framework for evaluating the obfuscation resilience of repackaging detection algorithms

Our study indicates that while each detection algorithm excels at detecting some types of repackaging efficiently and effectively, there is a lack of false negative analysis for all of them. Given the huge number of Android apps in different markets and the easiness in applying obfuscation techniques during repackaging, it becomes a very difficult problem to ensure both efficiency and accuracy of the repackaging detection algorithms. Therefore, we propose a general framework to help comprehensively evaluate the obfuscation resilience of such algorithms. The outcome of evaluation based on our framework can be used as a guidance to enhance the obfuscation-resilience of a detection algorithm through, for example, fine tuning of parameters and adding/removing certain heuristics.

The proposed framework is illustrated in Figure 3, which contains three major components. The first component is called *Dalvik Bytecode Pre-processor*, which dissembles and transforms the Dalvik EXecutable (DEX) into an intermediary representation (IR) code format. By using an IR format, code manipulation and obfuscation can be performed easily. The second component is an *IR Code Obfuscator*, which works directly on the output IR format of the original program. In the real world scenario, various obfuscators can be applied in different ways by plagiarists to evade the detection. By applying a set of obfuscation algorithms in various ways, the *IR Code Obfuscator* tries to mimic the real-world obfuscation based repackaging process. Hence, this component outputs a set of obfuscated versions from the original input dex file. During this process, we must ensure all the code manipulation and obfuscation actions are semantics-preserving transformations. After the obfuscation, the semantics equivalent IR code is converted back into Dalvik bytecode by *IR2Dex Repackager*, so that it is compatible with most detectors, which take dex files as input.

3.1 Dalvik bytecode pre-processor

The preprocessing phase performed on the Dalvik bytecode is to reverse it to an IR code format, so that it can be further manipulated easily and directly. Several intermediary representation candidates can be leveraged, including $smali$ format, Java bytecode format or other similar representation, like Jasmin. In our current framework design, we choose Java bytecode format as the IR, since a tool called $Dare$ [25] can directly translate the Dalvik bytecode into Java bytecode with a high success rate. In Android

platform, the type information for some specific Java bytecode instructions used in Java Virtual Machine is thrown away when *dx tool* compiles Java bytecode down to Dalvik bytecode in Android platform. However, the missed type information is inferred by $Dare$ using strong constraint solving and backward slicing. The experiment in $Dare$ shows that the information incompleteness between the converted Dalvik bytecode and Java bytecode is reduced to a tolerable rate. It achieves a total of 99% of verifiable code on average for thousands of tested methods from various Android apps. We will elaborate some interesting observations on using $Dare$ in our experiments in Section 4.

3.2 IR code obfuscator

This component is designed to mimic the real world scenarios where a plagiarist makes basic modifications or uses various obfuscation techniques for repackaging detection evasion. Since our motivation is trying to provide a general evaluation framework, we consider two pieces of information are very important to report. First is the $broadness$ analysis result, which shows the general weakness and strength for an evaluated detection algorithm against a broad range of obfuscation techniques. We decide to perform obfuscation algorithms individually and collect a set of detection results for the testing algorithm. This result will provide a wide range of analysis, to reveal the detection algorithm's strength and weakness against each type of obfuscation algorithm. The $broadness$ analysis result provides insights to improve the detection algorithm. Second is the $depth$ analysis result, which shows the overall obfuscation resilience against deep code manipulation by serializing a set of obfuscation techniques. In this analysis, the detection algorithm is tested against repackaged applications that have been obfuscated multiple times. For example, an application may be obfuscated by variable renaming, followed by noise injection and/or control-flow flattening. With depth analysis, we can test the robustness of detection algorithms against more sophisticated obfuscation attacks.

Specifically, because there already exist some comprehensive obfuscation tools, such as *e.g., SandMarks* [17] and *Zelix KlassMaster* [9], which target at Java bytecode, by leveraging them we can save some engineering effort by avoiding the re-implementation of a large set of obfuscation algorithms. The open source $SandMarks$ is a very comprehensive tool, which implements 39 obfuscation algorithms. $Klass\text{-}Master$ implements some control flow obfuscation techniques, making it a heavy duty obfuscator. We currently take both as our candidate obfuscators. However, our *IR code obfuscator* is not restricted to Java bytecode obfuscation tools. Whenever the obfuscation community provides better tools for Dalvik bytecode obfuscation or other similar IR format obfuscation tools, we will try to include them in our framework. Therefore, our framework can be incrementally updated for more comprehensive obfuscation resilience measurement.

3.3 IR2Dex repackager

In general, most detection algorithms take Dalvik bytecode of the Android program, namely the dex file, as the initial input, based on the assumption that no availability of source code. Therefore it is necessary to convert the obfuscated IR code back into the

dex file by the *IR2Dex Repackager* component. In this way, we can build a standard framework, which is compatible for most detection algorithms that target Dalvik bytecode. For this component, we currently leverage a solid tool named *dx tool* from the Android platform.

An important criteria for the repackager component is that it must preserve the effects of obfuscation performed on our IR code during the conversion. After analyzing the source code [7] of *dx tool*, we find *dx tool* provides very little optimization, except for some dead code removing and a few minor optimizations on register usage. We will further discuss our analysis on the behavior of *dx tool* in Section 4, which indicates that it satisfies the criteria for a repackager component.

3.4 Practical concern

To the best of our knowledge, currently no repackaging detection algorithm is based on dynamic analysis of Android applications. Due to the requirement of intensive user interactions through the touch-based UI for Android OS, it becomes very hard to automate the input feeding process for dynamic analysis of applications from a large-scale prospective. The other possible reason that static analysis is a better option is that a plagiarist can easily manipulate the GUI of an app and completely defeat dynamic based detection algorithms. Hence, in our current framework we do not aim at guaranteeing that obfuscated applications should be completely runnable. To make a repackaged application completely runnable is only a must requirement for the plagiarists, so that it can be published in the app market again. However, we relax this constraint in our evaluation framework so that we can provide a more comprehensive framework for static analysis based detectors. In other words, our framework indeed offers advantages to the attacker even if in reality he may not be able to apply these obfuscation algorithms easily. We consider the obfuscated $classes.dex$ file from our evaluation framework valid as long as it is the valid syntax of Dalvik bytecode, contains the equivalent program semantics of the original one, and can be accepted by Dalvik bytecode dissemblers and other static analysis tools. After all, if needed, our framework can be refined later to produce completely runnable apps for dynamic analysis based repackaging detection algorithms.

4 Experiment

In the framework setup, we choose $SandMarks$ as the obfuscator component, which has a comprehensive and powerful collection of obfuscation algorithms for Java bytecode manipulation and all of these obfuscation techniques are well documented. It meets the requirements from both the *broadness* and *depth* analysis aspects. Also, $Dare$ and *dx tool* are leveraged for bytecode reverse engineering and repackaging, respectively. To fully automate the experiment, we write shell-scripts to glue the command-line version of $SandMarks$ and other components, $Dare$ and *dx tool* together. Each component's output is fed into the next component as input and all the tasks in our framework are then pipe-lined. Thereafter, we test the success rate of producing obfuscated dex files under various obfuscations and also conduct one case study with an open source Android application repackaging detection tool, Androguard [20].

4.1 Framework setup

Preprocessing Android applications The preprocessing tool $Dare$ that we currently use is one of the most accurate Dex-to-Jar converter, as most of the converted code can be verified by the Oracle Labs Maxine VM verifier [10]; however, some ambiguous type inference problem cannot be completely solved even after performing the constraint solving algorithms for ambiguous type inference. We relieve this problem by two procedures. First, we turn on the *-c* option from $Dare$, which leverages the optimization provided by $Soot$ [13] for the reversed Java bytecode. This step can help remove the unnecessary Java bytecode and reduce the possibility of invalid bytecode instructions. Second, we relax the strict type checking performed during the $SandMarks$ preprocessing phase by modifying the latest source code of $SandMarks$. After analyzing its source code, we find the strict type checking is performed in $SandMarks$ preprocessing phase by using the BCEL libraries [15]. Therefore, after relaxing the type checking from BCEL libraries, we are able to make $SandMarks$ accept all the Jar files output from $Dare$ and run them through various obfuscation algorithms.

Applying obfuscation algorithms For the $broadness$ analysis, we try to perform all the 39 obfuscation algorithms in $SandMarks$ separately; however, some obfuscation algorithms can not proceed successfully. For example, *"Array Splitter", "Array Folder"* and *"Integer Array Splitter"* are not able to complete. After further analysis, we find it is because of the missing type information during the backward conversion from Dalvik bytecode to Java bytecode.Dalvik bytecode does not contain the type information for the array relevant instructions. For instance, in DVM, it generally uses *"aget and aput"* for both *int* and *float* arrays and *"aget-wide and aput-wide"* for both *long* and *double* arrays. However, when trying to convert them into Java bytecode, the above opcode *"aget and aput"* need to be type inferred and mapped into *"iaload, iastore, faload and fastore"* strictly. It is the same case for *"aget-wide and aput-wide"*, which should be mapped to *"laload, lastore, daload and dastore"* strictly. Even after $Dare$'s inference of the relevant ambiguous typing from program context by leveraging strong constraint solving and backward slicing, it is still not able to resolve type ambiguity cases completely. In our work we do not solve this problem, as we consider this is one of the fundamental limitation of the conversion from Dalvik bytecode into Java bytecode. Fortunately, this does not hurt other obfuscation algorithms.

Another problem appears when trying to perform the $depth$ analysis. Theoretically speaking, it is possible to perform a series of obfuscation algorithms on a program to obtain a deeply obfuscated program. However, some obfuscation algorithms can cause conflicts, due to the limitations of their underlining implementation. Instead of blindingly attempting various permutations of all the algorithms, we first group the obfuscation algorithms into different categories, including *Layout Obfuscation*, *Control Obfuscation* and *Data Obfuscation*, labeled in the score column of Table 4.2 as "L", "C" and "D", respectively. Then based on the result of $broadness$ analysis, we try the combination among the most effective individual obfuscation algorithms from each category, so as to maximize the overall effectiveness of multiple obfuscation techniques while minimizing the opportunity of conflicts and the experiment space.

Table 1. The successful output benchmark for the framework

Dare Preprocessor		SandMarks Obfuscator		Android dx tool	
input# 20.*dex*	output# 20 *.jar*	input# 20 *.jar*	output# 720 *.jar*	input# 720 *.jar*	output# 720 *.dex*
100%		92.5%		100%	
Total Successful Rate		92.5% = 100% * 92.5% * 100%			

Framework stability Currently, we have comprehensively tested 20 Android applications downloaded from Android Official Market with 36 (out of 39) obfuscation algorithms provided by $SandMarks$ and output 36*20 obfuscated *classes.dex* files. As discussed previously, three obfuscation algorithms that are relevant to array manipulation cannot be performed completely. Both the $Dare$ and *dx tool* components perform well, except for few ambiguous type information cases caused by information loss in the first pre-processing component. By gluing these three components together, we reach a total success rate of 92.5% for the tested apps, which demonstrates the stability of our framework on performing the $broadness$ analysis. On the other hand, when applying blindingly various combinations of the obfuscation algorithms , $SandMarks$ tends to throw errors. After our grouping of relevant obfuscation techniques, we are able to perform four interesting serialization of obfuscation with a high success rate. We will provide detailed observations in Section 4.2.

4.2 Case study

In order to test the effectiveness of our framework for evaluating obfuscation resilience, we perform a case study with $AndroGuard$ [20], which is an advanced Android application repackaging detection algorithm presented in Blackhat 2011. It can directly perform similarity comparison between a pair of $classes.dex$ files from different apps. AndroGuard describes the an Android application as regular expression string, which can capture the control flow structure of the program very efficiently and effectively. Then pair-wise comparisons on the method units between two similar applications are conducted by leveraging the similarity distance computation algorithm based on Normalized Compression Distance (NCD). Then based on the threshold specified in the algorithms, AndroGuard can identify method relevant metric, including *"new method", "diff method"* and *"match method"*. The final similarity score is derived based on the metric.

Applying single obfuscation algorithm We performed a $Broadness$ analysis on AndroGuard against all the obfuscation algorithms from $SandMarks$ thoroughly using our framework. This analysis is performed in a control way, as each time only one obfuscation is applied and results are collected. Therefore, we can pinpoint the exact weakness of the repackaging detection algorithm. In Table 4.2, the algorithm columns indicate the names of the obfuscation algorithms applied in our framework. For the score column, we first use $AndroGuard$'s detection algorithm to compute the similarity score of a pair of original/obfuscated $classes.dex$ files of each Android application.

Table 2. Average Similarity Score by AndroGuard for each Obfuscation Algorithm

Algorithm	Score	Algorithm	Score
Non-obfuscated	1.00 (L)		
Const Pool Reorder	.92 (L)	Split Classes	.94 (L)
Static Method Bodies	.88 (C)	Class Encrypter	.03 (D)
Method Merger	.65 (C)	Reorder Parameters	.92 (D)
Interleave Methods	.56 (C)	Promote Prim Reg	.92 (D)
Opaque Pred Insert	.92 (C)	Promote Prim Types	.93 (D)
Branch Inverter	.77 (C)	Bludgeon Signatures	.96 (D)
Rand Dead Code	.92 (C)	Objectify	.83 (D)
Class Splitter	.87 (C)	Publicize Fields	.91 (D)
Method Madness	.43 (C)	Field Assignment	.86 (D)
Simple Opaque Pred	.92 (C)	Variable Reassign	.85 (D)
Reorder Instructions	.89 (C)	ParamAlias	.92 (D)
Buggy Code	.67 (C)	Boolean Splitter	.85 (D)
Inliner	.89 (C)	String Encoder	.87 (D)
Branch Insert	.87 (C)	Overload Names	.91 (D)
Dynamic Inliner	.84 (C)	Duplicate Registers	.89 (D)
Irreducibility	.86 (C)	Rename Registers	.96 (D)
Opaque Branch Insert	.85 (C)	False Refactor	.95 (D)
Exception Branch	.81 (C)	Merge Local Int	.94 (D)

We then compute an average over all the score of the tested pairs of the 20 applications under the same obfuscation algorithm. Therefore, the average similarity score is a good measurement on the average performance of $AndroGuard$ against individual obfuscation attack.

Applying single obfuscation algorithm In order to check the effect of the Dex2Jar and Jar2Dex conversions on $AndroGuard$, we also use $AndroGuard$ to compute the similarity score for the original dex file and the *"non-obfuscated"* dex file that has only been processed by $Dare$ and $dx\ tool$.From Table 4.2, the entry *"Non-obfuscated"* has the corresponding similarity score "1.00" from $AndroGuard$, which means it is 100% similar (the range of the similarity score is [0, 1]). This indicates that the two-way conversions by $Dare$ and $dx\ tool$ can keep almost all the semantic information of the code. Based on the classification in Collberg et al. [18], this transformation can be categorized as $Layout\ Obfuscation$, we tag it as "L" in the corresponding score column, as it touches very little semantic content of the code. All the other scores are below "1.00", which demonstrates that all the other obfuscation algorithms have more or less effect on $AndroGuard$'s detection result.

The algorithms on the left side of the table have "C" tagged on the corresponding scores, because they belong to the *"control transformation"* category, which tries to obscure the control-flow of the code. The ones on the right side and tagged with "D" belong to *"data transformations"*, which obscure the data structure used in the source applications. Generally speaking, $AndroGuard$ has better resilience to data structure

based obfuscations, since it does not take the detail data dependency or data structure into account. However, the *"Class Encrypter"* obfuscator makes an exception in this category, as this obfuscation reduces the similarity score to ".03". By encrypting class files and decrypting them at runtime, the *"Class Encrypter"* can completely change the semantics of the string structure that $AndroGuard$ uses to represent the Dalvik bytecode.

The other obfuscation methods that have big impacts are *"Method Merger"*, *"Interleave Methods"*, *"Method Madness"* and *"Buggy Code"*. To figure out the reason, we analyze the source code of AndroGuard in the similarity comparison part. Basically, the algorithm computes the Control Flow Graph (CFG) within each method, and represents each CFG of each basic block by a predefined regular expression representation and takes this string representation of each method as the core feature. By leveraging Normalized Compression Distance (NCD) algorithm, they can aggregate the final score. Since all the above four obfuscation algorithms are relevant to basic block and control flow manipulation performed on the method granularity, they can reduce the chance of repackaged apps being detected by their similarity measurement algorithm. Actually, these obfuscation algorithms directly obscure the core feature that AndroGuard is trying to extract from the code for comparison and detection. From this result of the $broadness$ analysis, our framework is able to comprehensively measure the obfuscation resilience of the detection algorithm and also pinpoints its weakness.

Serializing multiple obfuscation algorithms Practically, especially when detection algorithms become more powerful, it is very possible that an attacker will try a combination of various obfuscation algorithms. Therefore, our framework also wants to mimic more complicated obfuscation behavior in the real world scenario. Besides the $broadness$ analysis performed on AndroGuard, we also perform advanced multiple-obfuscation by serializing several algorithms for $depth$ analysis. It is a deeper analysis process on the obfuscation resilience of detection algorithms.

We test various combinations of the effective individual obfuscation based on the result of $broadness$ analysis. When trying various permutations, only some of them can be performed successfully for the testing applications and the output obfuscated DEX files can be accepted by $AndroGuard$. We analyze four interesting cases below:

1. *[Method Merger ⇒ Method Madness ⇒ Interleave Methods]*
   Average Similarity Score and Obfuscation Time of 18 apps : 0.33 and 19 min;
2. *[Objectify ⇒ Method Merger ⇒ Method Madness]*
   Average Similarity Score and Obfuscation Time of 19 apps : 0.26 and 16 min;
3. *[Method Madness ⇒ Objectify ⇒ Variable Reassign]*
   Average Similarity Score and Obfuscation Time of 20 apps : 0.35 and 11 min;
4. *[Variable Reassign ⇒ Boolean Splitter ⇒ Objectify]*
   Average Similarity Score and Obfuscation Time of 20 apps : 0.80 and 6 min;

We record the average similarity score computed by AndroGuard and the average total time needed for the whole process of serializing three obfuscation algorithms. All the test cases reduce the average similarity scores to a point which is lower than any the

single obfuscation performed individually in the $broadness$ analysis. The average total time is the sum of the time for applying each obfuscation algorithm.

Case 1 leverages three heavy control transformation based obfuscations that target specifically at method level manipulations. We choose these top-three obfuscations based on the $broadness$ analysis result. This serialization further reduces the similarity score down to a low point 0.33, which results in high false negatives in $AndroGuard$. Generally, for $AndroGuard$, it will cause lots of false positives when setting the threshold below 0.5 for the similarity score. This deep serialized obfuscation process requires more time to perform, about 19 minutes on average for each application. Also there is 5% chance that the output dex file could not be accepted by $AndroGuard$, as 18 out of 20 can be successfully accepted by $AndroGuard$.

Case 2 is a serialization of one data transformation plus two control transformations. Based on the previous $broadness$ analysis report, we try several combinations with *"Class Encrypter"* as data transformation, and find no further decrease on the similarity score. As *"Class Encrypter"* already brings the similarity score to a very low level (0.03), the effects of other obfuscation algorithms cannot be directly reflected in the score. Note all other possible obfuscations must be applied before *"Class Encrypter"*, so that the actual effect of these obfuscations can be kept. *"Objectify"* is another top data transformation based obfuscation, and we combine it with two top obfuscation algorithms in the control transformation category, namely *"Method Merger"* and *"Method Madness"*. This combination reduces similarity score to 0.26, which is more effective than the combination of the three top control transformations in Case 1. This is an indication that combining various obfuscations from different categories can potentially produce more powerful obfuscations. Based on further analysis of the result of Case 2, we find that using the obfuscations selected from different categories can increase the number of methods that are considered not similar by $AndroGuard$ between the original and obfuscated dex files. It is because the manipulations from different obfuscation categories touch different parts of the code and produce the obfuscated methods with less chance of overlapping with the original.

Case 3 serializes one control plus two data transformations, which also indicates that obfuscations from different categories performs better. This two data obfuscations, *"Objectify"* and *"Variable Reassign"* are only at a 0.8 level score when performed separately; however, after our serialization with the *"Method Madness"* on control transformations, it performs well and even approximates the similarity score from case 1. The whole transformation time is reduced to nearly half of the time spent in case 2 for each application and we obtain a 100% success rate for output dex files by this serialized obfuscation. All the 20 heavily obfuscated dex files are accepted by $AndroGuard$ successfully. The last case indicates by purely using the data transformation based obfuscations. The average similarity score is further reduced but not as significant as case 2 and 3.

4.3 Discussions

Our experiment and case study demonstrate the effectiveness of our framework for providing a comprehensive and deep measurement on the obfuscation resilience aspect of a proposed repackaging detection algorithms. The $broadness$ analysis measures the

obfuscation resilience of the detection algorithm comprehensively and points out its exact strength and weakness. On the other hand, $depth$ analysis can further attack the detector under an advanced obfuscation scenario, which provides a better understanding of its overall obfuscation resilience. We believe the insight from both analyses is helpful to understand the detection algorithm and can serve as a guidance for its enhancement.

We use AndroGuard to perform the case study because it is currently the only publicly available tool. However, the result from the case study can be applicable to all the other detection algorithms. For example, the *"Class Encrypter"* obfuscation can probably have the same effect on other detection algorithms based on static code analysis. Because it dramatically changes the original bytecode by encryption and only dynamic decryption can help unpack the obfuscation. Hence, without adding special heuristic to prevent this obfuscation, all the static code analysis based detection becomes ineffective. The case study result shows that AndroGuard is not very resilient against control flow manipulation based obfuscation. However, we envision the opposite result will probably be generated when using our framework to test DNADroid.

Since the obfuscation algorithms are performed on the intermediary format, the Java bytecode, which are later converted into $classes.dex$ by the *dx tool* on instruction level granularity, one may wonder whether the obfuscation effect has been preserved. We randomly pick and manually analyze outputs of all the 36 obfuscation algorithms and the corresponding serialized ones. According to the corresponding explanations from $SandMarks$, the effect of most control flow obfuscation and data obfuscation algorithms are preserved. The obfuscator is based on semantic preserving obfuscations from $SandMarks$. Moreover, the dx tool keeps the program's semantic of the input Java bytecode, and converts the Java bytecode instructions into semantic equivalent Dalvik bytecode instructions based on the predefined transformation rules. Therefore, most of the obfuscation effect on our intermediary representation is preserved in the output $classes.dex$ file.

We also confirm that some class level obfuscations, *e.g.*, "Class Splitter" and "Split Classes", need some modifications in the *AndroidManifest.xml* file, so that relevant class information will be updated accordingly. We suggest users to simply turn them off when performing the evaluation, if their detection algorithms try to leverage the information in the *AndroidManifest.xml*. For those which do not need the information from *AndroidManifest.xml*, they can still obfuscate $classes.dex$ by using the obfuscations in class level.

5 Related Work

Android application reverse engineering and code manipulation. Since Dalvik bytecode contains more semantic information than machine code instructions, its reverse engineering and manipulation are also easier. Several tools, including $Dex2jar$ [4], ded [24] and $Dare$ [25], can transform Dalvik bytecode to Java bytecode. Based on the converted Java bytecode representation of dex files, many static analysis tools on Java bytecode can be applied, *e.g.*, $WALA$ [14] and $Soot$ [13]. In evaluation framework, we also leverage this convenience to deploy $SandMarks$ [17] and $KlassMaster$ [9] in our obfuscator component. $Smali/baksmali$ [12], the assembler/disassembler

of $classes.dex$ files, not only can reverse engineer Android applications but is able to repackage the modified $smali$ code back to Dalvik bytecode. Tools such as $apktools$ [1] integrate $Smali/baksmali$ to help modify an application, sign with another developer key, and repackage it back into an *apk* file.

To counter reverse engineering, Android developers use obfuscation tools frequently such as ProGuard [11], DexGuard [5] and dasho [3], to prevent the repackaging at the initial stage. These obfuscation techniques rely on Java source code, Our evaluation framework is trying to mimic the obfuscations performed by plagiarists, which is under the assumption that there is no accessibility to Java source code.

Repackaging techniques can be leveraged to provide protection mechanisms, if used in a proper way. Aurasium [27] reverse engineers and repackages the dex files to perform bytecode rewriting, so that protection code can be embedded into the Android apps to specify policy enforcement within user-level sandboxing. In article [6], "Junk byte injection", a x86 architecture well-known obfuscation technique, is proved to be applicable on Dalvik bytecode format to raise the bar of further malicious reverse engineering on Dalvik bytecode.

Repackaging detection and evaluation. Paper [30] analyzes the evolution of the Android malware and current status of the repackaging and obfuscation techniques that have been used. We perform study from another prospective, that is trying to analyze and measure the obfuscation resilience of repackaging detection algorithms in [29], [19], [23].

Wang et al., [26] design a system call based software birthmark that represents the unique characteristic of the run time behavior of a program, which can be used for software theft detection. They measure their birthmark against various obfuscations and also with different compiler setups. Jhi et al., [21] design a plagiarism detection technique, which is resilient to various control and data obfuscation techniques. The detection is based on an observation that some critical runtime values are hard to be replaced or eliminated by semantics preserving transformation techniques. They evaluate the obfuscation resilience of the value-based method through SandMark, KlassMaster, Thicket and Loco/Diablo.

The evaluation of the obfuscation techniques has been studied in [16], which assesses how difficulty it is for an attacker to understand and modify obfuscated code through controlled experiments involving human subjects. Karnick et al., [22] propose a standard measurement to analyze and evaluate the strength of obfuscation tools. An analytical metric is developed to quantify the performance of obfuscation in terms of potency, resilience, and cost. Our work provides a general framework to measure the obfuscation resilience of repackaging detection algorithms. We have evaluated our framework using a case study on a real repackaging detection algorithm.

6 Conclusion

Due to the improved code manipulation techniques of code manipulation on Dalvik bytecode, it is very important for repackaging detection algorithms to be obfuscation resilient, so that more stealthy repackaging scenarios can be identified. In this work,

we propose a framework to help evaluate the obfuscation resilience of detection algorithms in terms of broadness and depth. The framework provides a uniform obfuscation resilience measurement for all the obfuscation detection algorithms that are based on static analysis of Dalvik bytecode. Our experiments have demonstrated that our framework is stable to create obfuscated $classes.dex$ for the $broadness$ analysis and also is able to serialize multiple obfuscations together to perform the $depth$ analysis. Our study on the serialization of multiple obfuscations from different categories provides some understanding on how to make a stronger obfuscation. Our case study on $Androguard$, shows that our framework can effectively pinpoint the exact strength and weakness of the detection algorithm. The outcome of evaluation based on our framework can be used as a guidance to enhance the obfuscation-resilience of a detection algorithm through, for example, fine tuning of parameters and adding/removing certain heuristics.

Acknowledgments. We would like to give special thanks to Professor Christian Collberg for his help on using the SandMark tool and Damien Octeau's detail clarification on the Dare tool.

This work was partially supported by ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, NSF CNS-0916469, NSF CNS-1223710, AFOSR W911-NF1210055 from Liu, NSF CAREER 0643906 from Zhu and NSF Grant CNS-1223710 from Wu. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reect the views of the National Science Foundation, Army Research Office or AFOSR.

References

1. Android Apktool: A tool for reengineering android apk files. http://code.google.com/p/android-apktool/.
2. Dalvik virtual machine: code and documentation. http://code.google.com/p/dalvik/.
3. Dasho, preemptive solutions. http://www.preemptive.com/products/dasho.
4. Dex2jar. http://code.google.com/p/dex2jar/.
5. Dexguard. http://www.saikoa.com/dexguard.
6. Dexobf. http://dexlabs.org/blog/bytecode-obfuscation.
7. Dx tool source code. http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/4.1.2_r1/com/android/dx/ssa/.
8. Gartner says android to command nearly half of worldwide smartphone operating system market by year-end 2012. http://www.gartner.com/it/page.jsp?id=1622614.
9. Klassmaster. http://www.zelix.com/klassmaster/docs/index.html.
10. Oracle Virtual Machine. https://wikis.oracle.com/display/MaxineVM/Home/.
11. ProGuard. http://proguard.sourceforge.net/.
12. Smali/Baksmali. http://code.google.com/p/smali/.
13. Soot: a Java optimization framework. http://www.sable.mcgill.ca/soot/.
14. Wala. http://wala.sourceforge.net/wiki/index.php/.

15. Byte code engineering library (bcel). http://sourceforge.net/projects/javaclass/.
16. M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. Towards experimental evaluation of code obfuscation techniques. In *Proceedings of the 4th ACM workshop on Quality of protection*, QoP '08, pages 39–46, New York, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1456362.1456371.
17. C. Collberg, G. Myles, and A. Huntwork. Sandmarks a tool for software protection research. In *IEEE Security and Privacy, vol. 1, no. 4*, page 4049, 2003.
18. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, 1997.
19. J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *ESORICS*, pages 37–54, 2012.
20. A. Desnos and G. Gueguen. Android: From reversing to decompilation. In *Black hat 2011, Abu Dhabi*.
21. Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 756–765. ACM, 2011.
22. M. Karnick, J. Macbride, S. Mcginnis, Y. Tang, and R. Ramach. A qualitative analysis of Java obfuscation.
23. S. Li. Juxtapp: A scalable system for detecting code reuse among android applications. Master's thesis, EECS Department, University of California, Berkeley, May 2012. http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-111.html.
24. D. Octeau, W. Enck, and P. McDaniel. The ded Decompiler. Technical Report NAS-TR-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Sept. 2010. http://siis.cse.psu.edu/ded/papers/NAS-TR-0140-2010.pdf.
25. D. Octeau, S. Jha, and P. McDaniel. Retargeting Android Applications to Java Bytecode. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, November 2012. http://siis.cse.psu.edu/dare/papers/octeau-fse12.pdf.
26. X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 149–158. IEEE, 2009.
27. R. Xu, H. Saıdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security*, 2012.
28. I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *In Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, 2010.
29. W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.
30. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.