Reversing Obfuscated Control Flow Structures in Android Apps using ReDex Optimizer

Geunha You  
Dept. of Computer Science and Engineering, Dankook University  
Republic of Korea  
geunhayou@dankook.ac.kr

Gyoosik Kim  
Dept. of Convergence Technology, Korea Telecom  
Republic of Korea  
erewe2@gmail.com

Jihyeon Park  
Dept. of Software Science, Dankook University  
Republic of Korea  
fhfhwlgus@naver.com

Seong-je Cho  
Dept. of Computer Science and Engineering, Dankook University  
Republic of Korea  
sjcho@dankook.ac.kr

Minkyu Park  
Dept. of Software Technology, Konkuk University  
Republic of Korea  
minkyup@kku.ac.kr

ABSTRACT
Code obfuscation is a technique that makes programs harder to understand. Malware writers widely use the obfuscation technique to evade detection from anti-malware software, or to deter reverse engineering attempts for their malicious code. When we de-obfuscate the obfuscated code and restore it to the original code before obfuscation was applied, we can analyze the obfuscated malware effectively and efficiently. In this paper, we apply ReDex optimizer for reversing the control-flow obfuscation performed by the Obfuscapk system on open-source Android applications. We then analyze the effectiveness and limitations of ReDex in terms of its deobfuscation ability to reverse the control-flow obfuscation of Android apps. The experimental results show that ReDex can recover 1089 of 1108 apps obfuscated with control-flows obfuscation techniques of Obfuscapk obfuscator. During the process of optimizing bytecode, ReDex reduces the number of methods and fields significantly while it has a limitation in removing dead codes related to both useless goto statements and random nop instructions.

CCS CONCEPTS
• Security and privacy → Software reverse engineering, Malware and its mitigation. • Information systems → Open source software.

KEYWORDS
Android, Control-flow, Deobfuscation, Reverse Engineering.

1 INTRODUCTION
Numerous Android apps are distributed through the official Google Play app store or third-party online app stores. As Android apps become more widely used, the threat of malicious Android apps has increased significantly. According to the McAfee Mobile Threat Report in the first quarter of 2020, the total number of mobile malware in the fourth quarter of 2019 reached 37 million. This is an increase of about 16% compared to 32 million cases in the same period last year [11]. Therefore, the efficient detection of Android malware is very important. On the other hand, malware authors widely use some kinds of obfuscation techniques to evade detection from anti-malware software or to deter the attempts for reverse-engineering their malicious code [4, 13]. Code obfuscation is a technique that makes programs harder to understand while preserving the code semantics of the programs [2, 5, 13]. Typical obfuscation techniques applied to Android malicious apps include identifier renaming, string encryption, control-flow obfuscation, and Java reflection (API hiding).

To analyze the obfuscated malware effectively and efficiently, malware analysts need to de-obfuscate the obfuscated malicious code and restore it to the original code which was not transformed by an obfuscator. Well-known Android obfuscators include ProGuard [3, 13], and Obfuscapk [1]. ProGuard performs layout obfuscation (identifier renaming), and Obfuscapk supports several obfuscation techniques such as Align, Randomize Manifest, Renaming, Encryption, CallIndirection, Goto, Reorder, ArithmeticBranch, Reflection, etc. Especially, Obfuscapk performs various control-flow obfuscation by (1) adding new methods that invoke the original ones, (2) inserting a goto instruction pointing to the end of the method and another goto pointing to the instruction after the first goto, (3) adding some useless and semantic preserving instructions to the code, and (4) inserting random nop instructions.

In this paper, we apply ReDex bytecode optimizer [10] for reversing the control-flow obfuscation performed by the Obfuscapk [1] on open-source Android applications. ReDex is a software framework for reading, writing, analyzing, and optimizing Android bytecodes. The framework supports (1) minification and compression that reduces and minimizes strings size used by identifiers, class paths, and file paths, and (2) code inlining that moves functionality of a called method to the body of its caller. It also eliminates dead code (unreachable code). We analyze the effectiveness and limitations of ReDex in terms of its deobfuscation ability to reverse the control-flow obfuscation of Android apps performed by Obfuscapk.
2 BACKGROUND

2.1 Control-flow Obfuscation

In this paper, we consider only control-flow obfuscation of various code obfuscation techniques. Control-flow obfuscation aims to transform an app’s control flow so that the transformed control-flow has an intractable number of basic-block ordering combinations [2], [5].

Control-flow obfuscation commonly uses opaque predicates (or conditional instructions) that always return the same truth value, however, it is difficult to be statically determined. Inserting an opaque predicate to target code gives two branches: one for the original code, and another for unexecuted junk instructions. Another type of control-flow obfuscation is control-flow flattening. It divides a program into several code blocks and puts each code block as a case of a switch statement. It is not easy to determine the actual ordering of the code blocks using static analysis because the used index register is set based on an opaque value, which is hard to be statically evaluated.

Control-flow obfuscation makes code analysis difficult by adding dummy code/dead code, changing loop statements, inserting try-catch statements, etc. Therefore, the size of the DEX file increases in Android apps by control-flow obfuscation.

2.2 Obfuscapk

Obfuscapk is an open-source automatic obfuscation tool for Android apps [1]. It can be applied as a black-box obfuscation tool to benign and malicious apps for building or attacking a machine learning model. Obfuscapk can obfuscate compiled Android apps, and support various obfuscation techniques which are divided into two main categories: trivial and non-trivial techniques. Trivial techniques include Align, Re-sign, Rebuild, and Randomize Manifest, and can trick signature-based anti-malware tools.

Non-trivial techniques are more complicated and are applied to both bytecode and resources. They can be classified into four subcategories: Identifier renaming, Encryption, Code, and Reflection. The identifier renaming substitutes each identifier (package/classes/method names and variable names) with an opaque one. The encryption techniques can encrypt Native libraries, Strings in the strings.xml resource file, Constant strings in the code, and Asset files (e.g., videos, photos, etc.) using a random secret key. The code subcategory includes all transformation techniques that affect instructions in the classes.dex of an APK file. The obfuscation techniques provided by Obfuscapk are as follows: (1) removing debug meta-data (line numbers, types, or method names), (2) modifying the control-flow graph (adding new methods that invoke the original ones), (3) inserting a goto instruction pointing to the end of the method and another goto pointing to the instruction after the first goto, (4) changing the order of basic blocks in the code, (4) adding some useless and semantic-preserving instructions to the code, (5) inserting random nop instructions within every method, and (6) exploiting the overloading feature of the Java language to assign the same name to different methods but using different arguments. The obfuscation techniques are shown in Table 1. The reflection allows examining or modifying the run-time behavior of a class during execution. In this paper, we focus only on the code subcategory of non-trivial obfuscation techniques.

Table 1: Obfuscation techniques of Obfuscapk

<table>
<thead>
<tr>
<th>Category</th>
<th>Obfuscapk obfuscator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivial</td>
<td>RandomManifest, Rebuild, NewAlignMent, NewSignature</td>
</tr>
<tr>
<td>Renaming</td>
<td>ClassRename, FieldRename, MethodRename</td>
</tr>
<tr>
<td>Encryption</td>
<td>LibEncryption, ResStringEncryption, AssetEncryption, ConstStringEncryption</td>
</tr>
<tr>
<td>Code</td>
<td>ArithmeticBranch, Reorder, CallIndirection, DebugRemoval, Goto, MethodOverload, Nop</td>
</tr>
<tr>
<td>Reflection</td>
<td>Reflection, AdvancedReflection</td>
</tr>
</tbody>
</table>

2.3 ReDex optimizer

An Android bytecode optimizer, ReDex [8, 10], provides a framework for reading, writing, and analyzing DEX files, and various optimization passes that uses this framework to improve the bytecode. One of the optimizations performed by ReDex is to remove interfaces that have only one implementation. An APK optimized by ReDex should be smaller and faster than the original APK.

By performing optimization at the bytecode level, ReDex provides maximum ability to do global, interclass optimization across the entire executable file. It performs the transform on DEX bytecode rather than Java bytecodes. This process is similar to the post-linking stage in a C-style compilation process, where global optimizations are applied across the entire binary.

The optimization process of ReDex is shown in Figure 1. The optimization pipeline is organized as a series of states, with the “original” DEX entering at the beginning of the pipeline and the “transformed” DEX exiting at the end. Each stage in the pipeline can be considered as a stand-alone “optimization plugin”. The pipeline has the following stages (or transformation plugin): minification and compression, inlining, and dead code elimination. Each transformation in the pipeline is independent of each other.

Wermke et al. [13] considered ReDex as one of Android obfuscators. They described that ReDex accepted ProGuard’s configuration files and mirrors the renaming functionality closely. In this paper, we consider ReDex as one of Android deobfuscators because it can perform minification, inlining, and dead code elimination. That is, control-flow obfuscation can be reversed by ReDex.

Figure 1: Optimization process of ReDex

3 EXPERIMENTS AND EVALUATION

3.1 Experimental environment and Dataset

For experimentation and evaluation, an obfuscated dataset was created by applying the control flow obfuscation option provided by Obfuscapk [10] to the dataset of F-droid [7]. F-droid distributes 1426 open source APKs, and we randomly select 10 of them to see if
they can be installed and run on AVD (Android Virtual Device). We applied Obfuscapk’s control flow obfuscation technique to these 10 apps to create obfuscated APKs. After that, it was confirmed that the 10 obfuscated apps could also be installed and executed on AVD.

The experimental environment is as follows:
- OS: Ubuntu 18.04 LTS
- Android Virtual Device: Oreo 8.1 on Google Pixel 2 XL
- Decompiler & Disassembler: JEB Pro 3.24

3.2 Experimental results

We reverse 10 obfuscated apps using ReDex and confirm the de-obfuscated files can be installed and executed on AVD. In Table 2, we present DEX file sizes of the original, the obfuscated, and the optimized version of these 10 apps. Also, Table 3 shows the size ratio of the original to the obfuscated, the optimized to the original one, and the optimized to the obfuscated one, respectively.

Table 2: Original, obfuscated, and optimized DEX size (in bytes)

<table>
<thead>
<tr>
<th>Apps (.apk omitted)</th>
<th>Original</th>
<th>Obfuscated</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>An.stop_9</td>
<td>45,072</td>
<td>192,620</td>
<td>15,336</td>
</tr>
<tr>
<td>com.github.redpanal.roid_2</td>
<td>477,396</td>
<td>501,200</td>
<td>424,468</td>
</tr>
<tr>
<td>com.hobbyone.HashDroid_d_20</td>
<td>166,728</td>
<td>559,488</td>
<td>520,284</td>
</tr>
<tr>
<td>com.majeur.applications.info_7</td>
<td>85,584</td>
<td>347,352</td>
<td>298,008</td>
</tr>
<tr>
<td>com.oakley.fon_152</td>
<td>64,836</td>
<td>226,248</td>
<td>185,416</td>
</tr>
<tr>
<td>com.pierreduchemin.pun.chlinebingo_6 de.cweiske.headphone.indicator_1</td>
<td>1,675,088</td>
<td>1,852,616</td>
<td>1,607,012</td>
</tr>
<tr>
<td>de.duenndns.gmdice_6</td>
<td>3,876,692</td>
<td>4,066,840</td>
<td>3,712,128</td>
</tr>
<tr>
<td>de.cweiske.headphone.indicator_1</td>
<td>3,162,616</td>
<td>3,673,676</td>
<td>3,235,456</td>
</tr>
<tr>
<td>de.duenndns.gmdice_6</td>
<td>355.44</td>
<td>296.17</td>
<td>83.33</td>
</tr>
<tr>
<td>org.notabug.lifeuser_moves.iadb_190</td>
<td>116.16</td>
<td>102.30</td>
<td>88.07</td>
</tr>
<tr>
<td>Average</td>
<td>253.27</td>
<td>186.15</td>
<td>79.35</td>
</tr>
</tbody>
</table>

The DEX file size of the obfuscated version is about 1.05 times to 4.27 times the original dex size (MIN: grmpl.mk.stepandheightcounter_5.apk, MAX: An.stop_9.apk). The size of the optimized version is about 0.34 times to 3.48 times the original version (MIN: An.stop_9.apk, MAX: com.majeur.applicationsinfo_7.apk). The size of the optimized version is about 0.08 times to about 0.93 times the size of the obfuscated version (MIN: An.stop_9.apk, MAX: com.hobbyone.HashDroid_d_20.apk). On average the obfuscated version is about 2.53 times larger and the optimized version about 1.86 times larger than the original version. The optimized version is about 0.79 times the obfuscated version.

Table 3: The ratio of sizes of the original, obfuscated, and optimized DEX files (%)

<table>
<thead>
<tr>
<th>Apps (.apk omitted)</th>
<th>Obfuscated/original</th>
<th>Optimized/original</th>
<th>Optimized/obfuscated</th>
</tr>
</thead>
<tbody>
<tr>
<td>An.stop_9</td>
<td>427.36</td>
<td>34.03</td>
<td>7.96</td>
</tr>
<tr>
<td>com.github.redpanal.roid_2</td>
<td>104.99</td>
<td>88.91</td>
<td>84.69</td>
</tr>
<tr>
<td>com.hobbyone.HashDroid_d_20</td>
<td>335.57</td>
<td>312.06</td>
<td>92.99</td>
</tr>
<tr>
<td>com.majeur.applications.info_7</td>
<td>405.86</td>
<td>348.21</td>
<td>85.79</td>
</tr>
<tr>
<td>com.oakley.fon_152</td>
<td>348.95</td>
<td>285.98</td>
<td>81.95</td>
</tr>
<tr>
<td>com.pierreduchemin.pun.chlinebingo_6 de.cweiske.headphone.indicator_1</td>
<td>110.60</td>
<td>95.94</td>
<td>86.74</td>
</tr>
<tr>
<td>de.duenndns.gmdice_6</td>
<td>222.85</td>
<td>202.18</td>
<td>90.72</td>
</tr>
<tr>
<td>grmpl.mk.stepandheightcounter_5</td>
<td>355.44</td>
<td>296.17</td>
<td>83.33</td>
</tr>
<tr>
<td>org.notabug.lifeuser_moves.iadb_190</td>
<td>104.90</td>
<td>95.76</td>
<td>91.28</td>
</tr>
<tr>
<td>Average</td>
<td>253.27</td>
<td>186.15</td>
<td>79.35</td>
</tr>
</tbody>
</table>

The number of methods increased by about 7.6 times compared to the original. Also, it was found that the number of methods of the optimized version is about 0.64 and 4.89 times compared to that of the obfuscated and original version, respectively. In the case of another app com.majeur.applicationsinfo_7.apk, the optimized version has about 0.71 times the number of methods compared to the obfuscated version. That is, ReDex can reduce the number of methods and fields in bytecode obfuscated with control-flow obfuscation techniques.

Table 4: The number of classes, methods, and fields of the original, obfuscated, and optimized An.stop_9.apk

<table>
<thead>
<tr>
<th>Apps (.apk omitted)</th>
<th>The number of classes</th>
<th>The number of methods</th>
<th>The number of fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>An.stop_9.apk</td>
<td>34</td>
<td>133</td>
<td>230</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>36</td>
<td>1015</td>
<td>232</td>
</tr>
<tr>
<td>Optimized</td>
<td>35</td>
<td>650</td>
<td>97</td>
</tr>
</tbody>
</table>

Figure 2(a), (b), and (c) show the source codes of a method in the original, obfuscated, and optimized versions, respectively. All are simple forms composed of one statement in source code, but the difference is clear. In the case of obfuscated and optimized versions, a method that cannot infer the functionality from the method name only (e.g. AnstopDbAdapter.YUsHWPhDndlcfTpz) is called, and this method is a method created by Obfuscapk’s CallIndirection obfuscation option to complicate the control flow. In the case of an obfuscated version, several methods with meaningless names are used (Figure 2(b)). In the optimized version of Figure 2(c), you can see that ReDex effectively simplified the complex methods obfuscated by the Obfuscapk tool.

However, ReDex did not completely remove the calls of the method created by CallIndirection. Besides, it was unable to remove meaningless goto statements and random nop instructions added.
4 RELATED WORK

ProGuard [6] is one of the most well-known obfuscation tools for Android apps. As part of the Android SDK, the ProGuard applies identifier renaming for obfuscation. Recently, Android Gradle plugin 3.4.0 or higher no longer uses ProGuard to perform compile-time code optimization. Instead, the plugin works with the R8 compiler to handle code shrinking, resource shrinking, obfuscation, and optimization. The obfuscation technique supported by R8 compiler shorten the name of classes and members.

DeGuard [4] is a system to reverse obfuscations performed by ProGuard. It is a deobfuscation tool to (i) reverse the layout obfuscation performed by the ProGuard, (ii) rename obfuscated program elements of Android malware, and (iii) predict third-party libraries imported by benign APKs (also obfuscated by ProGuard). DeGuard could recover 79.1% of the program elements such as class, method, and variable names which were obfuscated with ProGuard.

Baumann et al. [3] developed the Anti-ProGuard tool that examines code parts in its database, which are similar to the code contained in the app, and use an established similarity relationship to suggest meaningful names to its packages, classes, and methods. Anti-ProGuard accepts an APK file obfuscated with identifier renaming techniques of ProGuard as an input and produces the deobfuscated version as an output. Anti-ProGuard could identify over 50% of known packages of Android apps. DeGuard and Anti-ProGuard focused on considering identifier renaming obfuscation but did not take into account control-flow obfuscation.

Moses and Mordekhay [9] implemented a deobfuscation solution that dealt with two obfuscation techniques: string encryption and dynamic method binding by reflection. The solution is a complex process that was implemented in which the dynamic analysis module passed the extracted data onto the static module. As a result, the authors have successfully deobfuscated the Android apps obfuscated by DashO. Since dynamic analysis cannot guarantee full code coverage, their approach may not deobfuscate all the obfuscated strings and dynamic method binding. They did not consider control-flow obfuscation techniques either.

Su et al. [12] proposed an approach to deobfuscating Android apps that leverages deep learning and topic modeling on machine code, MACNETO. Previous approaches to automatic deobfuscation have relied on certain structural parts of an app that remain unaffected by obfuscation. MACNETO makes almost no assumptions about the types of transformations and automatically deobfuscates the program by using topic modeling for instructions and deep learning techniques. The authors have shown high precision against ProGuard and Allatori.

5 CONCLUSIONS AND FUTURE WORK

ReDex is an open-source tool that optimizes Android apps. ReDex can reverse control-flow obfuscations. In this paper, we evaluated the performance of ReDex by optimizing the ten Android apps to which control flow obfuscation was applied. Experimental results have shown that ReDex reduced the number of classes and methods in the Android apps transformed with several control-flow obfuscation techniques of Obfuscapk obfuscator. Therefore, ReDex can help developers to implement automated malware analysis solutions which can tackle control-flow obfuscation techniques efficiently in Android.

Experimental results have shown that ReDex cannot remove the meaninglessly inserted goto and nop instructions. This implies that ReDex needs to be improved for dealing with useless goto and nop instructions. Thus we plan to develop a more effective tool for reversing control-flow obfuscation which can overcome those limitations of ReDex. Another possible direction for further research is to develop a tool that can handle other obfuscation techniques such as string encryption and Java reflections, as well as control-flow obfuscations. In addition, we will compare ReDex with the existing deobfuscation tools such as DeGuard and Anti-ProGuard.
ACKNOWLEDGMENTS
This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science and ICT (no. 2018R1A2B2004830) and the MSTI(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2020-0-00363) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation).

REFERENCES