Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Control flow obfuscation for Android applications

*Vivek Balachandran, Sufatrio *, Darell J.J. Tan, Vrizlynn L.L. Thing*

Institute for Infocomm Research, Singapore

ARTICLE INFO

Article history:
Received 16 December 2015
Received in revised form 11 March 2016
Accepted 12 May 2016
Available online 19 May 2016

Keywords:
Android
Software obfuscation
Mobile security
Reverse engineering
Application security

ABSTRACT

Android apps are vulnerable to reverse engineering, which makes app tampering and repackaging relatively easy. While obfuscation is widely known to make reverse engineering harder, complex and effective control flow obfuscations by rearranging Android bytecode instructions have not been implemented in various Android obfuscation tools. This paper presents our control-flow obfuscation techniques for Android apps at the Dalvik bytecode level. Our three proposed schemes go beyond simple control-flow transformations employed by existing Android obfuscators, and make it difficult for static analysis to determine the actual app control flows. To realize this, we also address a previously-unsolved register-type conflict problem that can be raised by the verifier module of the Android runtime system by means of a type separation technique. Our analysis and experimentation show that the schemes can offer effective obfuscation with reasonable performance and size overheads. Combined with the existing data and layout obfuscation techniques, our schemes can offer attractive measures to hinder reverse engineering and code analysis on Android apps, and help safeguard Android app developers' heavy investment in their apps.

1. Introduction

Recent years have seen a widespread and pervasive use of smartphones and tablet devices. The number of smartphones sold in 2015 surpassed 1.4 billion units, with Android dominating the market with an 80.7% share in Q4 (Gartner, 2013).

This extensive user base of Android devices and the alarming level of security threats targeting them make the security of Android applications (henceforth called *apps*) become a high priority (Sufatrio et al., 2015). One of the threats to Android apps is the relatively easy recovery of their app code by an attacker (Nolan, 2012). Android apps can be reverse engineered easily using off-the-shelf tools like apktool (apktool) or IDAPro (IDAPro), which results in rampant app tampering and repackaging (Zhou et al., 2012). This significantly threatens the fast growing mobile app market, which is estimated to value at $5.5 billions in 2015 and will reach $8.9 billions in 2018 (Schadler).

One of the techniques used against reverse engineering attacks is code obfuscation (Collberg et al., 1997). Obfuscation is the process of obscuring a piece of code so that it is harder to understand the reverse engineered code. Yet, the transformed code will not change the semantics of the original code. Due to this property, obfuscation has been recognized as a cost-effective mechanism to help app developers protect their released apps (Collberg and Thomborson, 2002).

* *Corresponding author.*
E-mail addresses: sufatrio@i2r.a-star.edu.sg (Sufatrio), balav@i2r.a-star.edu.sg (V. Balachandran), jjdtan@i2r.a-star.edu.sg (D.J.J. Tan), vriz@i2r.a-star.edu.sg (V.L.L. Thing).
http://dx.doi.org/10.1016/j.cose.2016.05.003
0167-4048/

Table 1 – Android obfuscators and the types of obfuscations performed.

Tools	Type of obfuscation								
	1	2	3	4	5	6	7	8	9
ProGuard (ProGuard)	√	√	√						√
DexGuard (DexGuard)	√	√	√	√	√	√	√		√
DashO (DashO)	√		√	√				√	
DexProtector (DexProtector)				√	√	√			
jarg (jarg)	√								
JODE (JODE)	√								√
Allatori (Allatori)	√			√				√	√
yGuard (yGuard)	√								

Existing Android obfuscation tools, however, seem to still lack complex control-flow obfuscation techniques (see Section 2).

This paper presents three schemes of performing control-flow obfuscation on Android apps at the Dalvik bytecode level, which go beyond simple control-flow transformations employed by existing Android obfuscators (see Section 3). The schemes disturb the normal control flow of an app method by making use of packed-switch construct, try-catch construct, and the combination of the two constructs. They make it difficult for static analysis to determine the actual control flows of the apps since the scattered code-block continuations and exception-raising operations are hard to be statically determined. While similar notion of control-flow flattening and signaling exist in the obfuscation literature (Chow et al., 2001; Popov et al., 2007; Wang et al., 2000), applying such techniques to Android bytecode still requires one to solve a possible *register-type conflict* issue, which can be raised by the Android runtime system[1] when an app's Dalvik bytecode is split, relocated and/or linked. To address this problem, we propose a *register-type separation* technique, which prevents the Android runtime system from encountering type conflict situations in the obfuscated apps (see Section 4).

Our analysis and experimentation show that the proposed schemes offer effective obfuscation with acceptable overheads (see Section 5). Experiments using popular apps and known benchmark suites show that our obfuscation incurs a reasonable size-overhead factor of 1.29, and a performance-overhead factor of 1.19. Combined with other existing forms of data and layout obfuscation techniques (Collberg et al., 1997), our proposed control-flow obfuscation schemes can therefore make reverse engineering attacks on Android apps much harder. Furthermore, our register-type separation technique will also enable other forms of control-flow obfuscation techniques to be applied on Android apps. As such, our proposed schemes will help hinder reverse engineering of Android apps, and safeguard app developers' heavy investment in their apps.

The remainder of this paper is organized as follows. Section 2 provides some background on Android obfuscation and runtime environments, and also compares related work. Section 3 presents our control-flow obfuscation schemes, while Section 4 describes the register-type conflict problem and elaborates our solution to it. Section 5 gives evaluation results of our obfuscation schemes. Section 6 further discusses our schemes' usability and limitations. Finally, Section 7 concludes this paper.

2. Background and related work

2.1. *Background on Android obfuscation*

Software obfuscation is a well-known technique used for protecting software from reverse engineering attacks (Collberg et al., 1997). Although it is theoretically impossible to achieve perfect obfuscation (Barak et al., 2012), obfuscating a program, in practice, makes its reverse engineering and code analysis harder (Collberg and Thomborson, 2002). Harrison (2015) investigates the effect of obfuscation on reverse engineering of Android apps. It concludes that obfuscation, even with basic techniques employed by ProGuard (ProGuard), can significantly increase the effort required to understand the obfuscated app. The importance of Android app obfuscation can be seen with the integration of ProGuard into the Android build system. The Android developers' documentation recommends releasing apps after obfuscating with ProGuard.

Android apps are released as Dalvik bytecode, which is then executed by the Android runtime system (more on this in Section 2.2). To analyze or manipulate an app in the absence of its source code, an attacker transforms the app bytecode into an assembly or higher level code representation. The motivation for the transformation may vary, including understanding the logic of a function, removing any watermark, or illegal app repackaging. Dalvik bytecode is an easy target for reverse engineering with various available reverse engineering tools, such as androguard (androguard), baksmali (smali/baksmali), apktool (apktool), IDAPro (IDAPro), dexdump (Android, Tools help) and dex2jar (dex2jar).

Obfuscation techniques that have been applied to Android apps by existing obfuscation systems include the following (see also Table 1):

1. Identifier renaming: It renames meaningful class and method names with arbitrary names like "a" and "b".
2. Class repackaging: This technique flattens existing multi-level class hierarchy by moving all classes into a single-level hierarchy.
3. Excessive overloading: It aggressively overloads different methods and fields using the same name.

[1] Android has two runtime virtual machines, namely Dalvik Virtual Machine (DVM) and the newer Android runtime (ART) (Android Open Source Project, ART and Dalvik). Section 2.2 gives some background on them and their differences.