Shielding Android Application Against Reverse Engineering

Sudipta Ghosh
MTech Scholar

S. R. Tandan
Assistant Professor

Kamlesh Lahre
Assistant Professor

Department of Computer Science & Engineering
Dr.C.V.Raman University, Bilaspur, India

Abstract

Abstract: In this growing market of smart phones, Android is an open source platform of Google that has become one of the most popular operating system. As a result, protecting applications running on Android becomes of interest. Currently, Reverse engineering of Android applications is much easier than on other architectures, due to the high level but simple byte code language used. Hence, the goal is to minimize and manage risks of software flaws. Anti-Reverse engineering techniques can be used to prevent reverse engineering. This paper discusses the possible code obfuscation techniques on the android platform. Our approach aims at increasing the complexity of the control flow of the application so that it becomes tough for a reverse engineer to get the business logic performed by an android application.

Keywords: *android; reverse engineering; code obfuscation; anti-reverse engineering; proguard*

1. Introduction

Reverse Engineering is the process of discovering the technical principles of a device, object or system through analysis of its structure, function and operation. Software Reverse Engineering is a process of analyzing the software structure, its behavior and to know more about how it works and operates. Reverse engineering can be used as learning purpose to learn how a software works. It can also be used for creating a new competitive software in a low price by understanding the behavior, structure of an existing software. Programs are written in a language, say C++ or Java, that's understandable by other programmers. But to run on a computer, they have to be translated by another program, called a compiler, into the ones and zeros of machine language. By decompiling that byte code one can get the source code of a program. By software reverse engineering one can know the technology working behind a software and can design a new software by taking some ideas from the existing software. Software reverse engineering is often an important part of the scientific method and technological development. The ideal result of a reverse engineering process for an Android application would be to reconstruct the original Java source code out of the distributed binary form. Obfuscation cannot prevent reverse engineering but can make it harder and more time consuming. We will discuss which obfuscation and code protection methods are applicable under Android.

“To reverse engineer a software application it is first necessary to gain physical access to it” (Low, 1998). The process of reverse engineering consists of three steps: (i) Parsing and semantic analysis of code, (ii)

Extracting information from the code, and (iii) Dividing the product into components, as indicated by Figure.1.

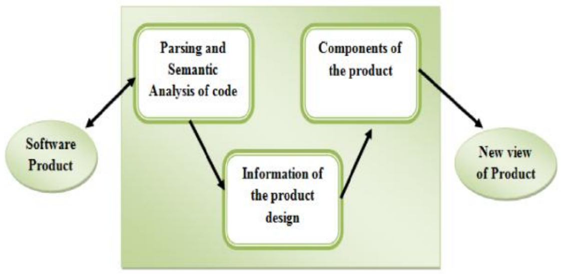

Software Product
Parsing and Semantic Analysis of code
Information of the product design
Components of the product
New view of Product

Figure.1 *Process of Reverse Engineering*

The Android [4] platform developed by the "Android Open Source Project" is one of the most popular systems for mobile devices in the recent years. This platform is designed in a way that the user can download and install new applications easily. Due to the popularity of the platform, the market for Android applications has grown massively both in variety and financial volume. This results in an increasing interest to protect program code of Android applications. This protection shall help against software piracy and serve as a method to guard intellectual property. On the Android platform there is a further motivation to protect the program code. It is not unlikely for a malware developer to abuse existing applications by injection of malicious functionalities and consequent redistribution of the trojanized versions [5]. Application developers are interested in protecting their applications. Protection in this case means that it should be hard to understand what an application is doing and how its functionalities are implemented.

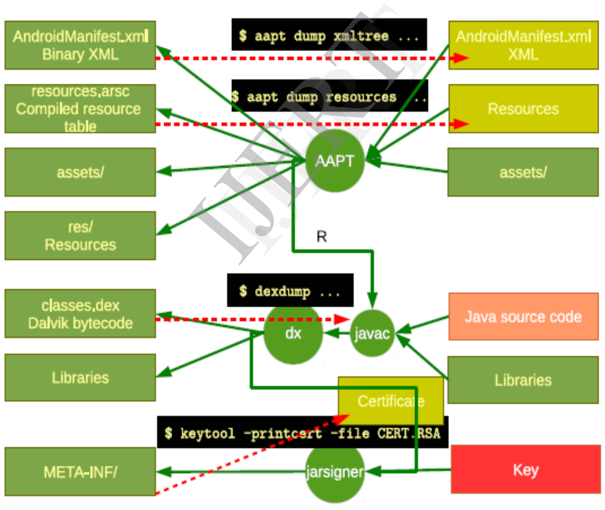

```
$ aapt dump xmltree ...
$ aapt dump resources ...
$ dexdump ...
$ keytool -printcert -file CERT.RSA
```

Figure 2. *APK- Android Package*

An Android application needs many steps and tools until the Android Application Package (APK) is build and ready to be deployed. The first step of the build process, as shown in figure 3 step 1, starts with the Java files which will be compiled into ".class" files by a Java compiler. The next step is the transformation from Java bytecode into Dalvik bytecode. For this, the "dx" program is used in step 2. It is included in the Android Software Development Kit (SDK) due to it is necessity for building an application for the Android platform. The output of dx will be saved into a single dex file "classes.dex". This file will be included in an APK in a later step. The last step of the build process is packing and signing the APK. The ApkBuilder constructs an apk file out of the "classes.dex" file and adds further resources like images and ".so" files. ".so" files are shared objects which contains native functions that can be called from within the DVM. The "jarsigner" just adds the developer signature to the APK, which can now be installed on an Android device.

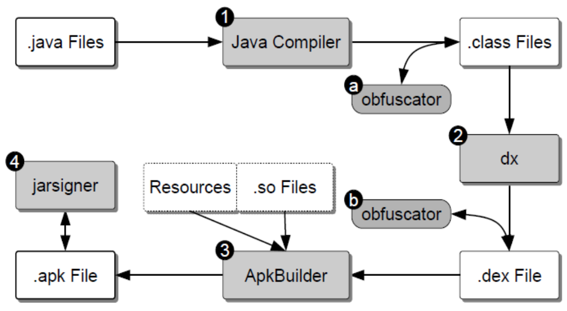


Figure 3. *Android application build process*

Obfuscation[1] is the paradigm of hiding program semantics through choosing semantically equivalent but complex and ambiguous representations in order to aggravate analysis. In order to achieve this protection, obfuscation transforms program code of an application in a way that it is "hard to reverse engineer" but without changing the behaviour of this application. Hard to reverse engineer means that automated programs cannot produce good abstractions of the analyzed program and the results of the analysis become harder to understand for a human analyst.

Anti-reversing techniques [3] are defence techniques implemented in software in order to protect it from malicious attacks. It has become a challenge for the software community to protect software from attackers and to prevent its misuse. The patent system is not quite as effective with software as it is with traditionally engineered tangible artifacts. While a patent mandates Intellectual Property (IP) protection – it is next to impossible to prove or even suspect any IP theft in a software product that might have been the result of a malicious reverse engineering attack on a patented competitor. A lot of research is being done in the software field in order to find out successful ways of protecting software from reverse engineering attacks. The techniques proposed to make reverse engineering difficult include obfuscating the code protecting the computing platform physically, encryption of executables, and watermarking.
The mentioned tool can be used within the deployment process of an application to obfuscate program code and protect the application against analysis.

1.1 ProGuard

ProGuard [11] is an open source tool which is also integrated in the Android SDK [4]. It can be easily used within the development process. ProGuard is basically a Java obfuscator but can also be used for Android applications because they are usually written in Java. The feature set includes identifier obfuscation for packages, classes, methods. Besides these protection mechanisms it can also identify and highlight dead code so it can be removed in a second, manual step. Unused classes can be removed automatically by ProGuard.

2. Related Work

Researchers have proposed various anti-reversing techniques to prevent reverse engineering. Currently there are so many techniques available but none of them provides 100% protection against reverse engineering. “Code Protection in Android by Patrick Schulz”[1] discusses the possible code obfuscation methods on the Android platform using Identifier mangling ,String obfuscation , Dynamic code loading , dead code, Self modifying code.

Identifier mangling

Identifiers are names for packages, classes, methods, and fields. They are represented as strings. In figure.4, a snippet of Java source code with highlighted identifiers is shown.

```
public class NetworkManager {
 private String encrypt( String input )
 { ... }
 public void send( String input )
 { ... }
}
```

Figure 4. *Java source code example with highlighted identifiers.*

In the example it is easy to get an idea of what this class is about and that it is almost certain that some kind of encryption is involved. From the example it is obvious, that original identifiers give information about interesting parts of a program. Reverse engineering methods can use these information to reduce the amount of program code that has to be manually analyzed. Identifier mangling aims to neutralizing these information in order to prevent this reduction.

```
public class a {
 private String a( String ab )
 { ... }
 public void b( String ac )
 { ... }
}
```

Figure 5. *Java source code example with obfuscated identifiers.*

String obfuscation.

Strings are arrays of characters which are frequently used within a program e.g. for enabling user interaction or printing messages. The original content of a string must be available at runtime because a user cannot understand an obfuscated or encrypted message dialog. In figure 6 an example of string usage is shown.

```
private void start() {
 String server = "irc.cnc.mal";
 String password = "notpublic";
 connection.connect( server , password );
}
```

Figure 6. *Java source code example with highlighted string*

String obfuscation can be achieved by any injective function F which is invertible and transforms an arbitrary string into another string.

Dynamic code loading

The obfuscator has to generate two components, the encrypted application and a decrypter stub. In Android, the encrypted application would be an encrypted dex file. This is rather easy to generate compared to the decrypter stub. The decrypter stub has to implement four main functionalities as shown in figure 7.

At first the encrypted application has to be fetched into memory fetched. This can be done by downloading a dex file from a remote server or extracting it from an internal data structure. In Android we can simple add
arbitrary files to an APK and access them at runtime.

The second step in figure 7 is the decryption of the encrypted dex file, yielding the original dex file. The cryptographic function can be chosen freely, due to its application is completely independent from the content of the dex file. After the unpacking stub has generated the original dex file, it can be loaded into the DVM and executed, as shown in steps 3 and 4 in figure 7.

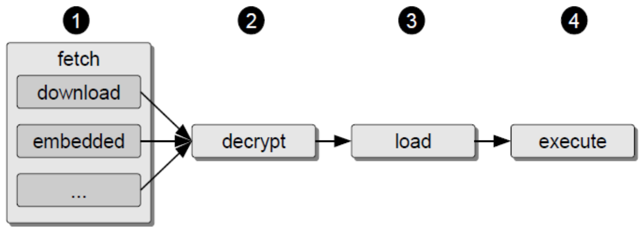


Figure 7. *The steps of an decryption stub in case of a packed application*.

To circumvent the restriction we can use the provided "Java Native Interface" (JNI) of the DVM, see figure 8.
JNI is intended to allow execution of native code, which is located in shared objects, out of the DVM. This is useful e.g. for computationally complex algorithms like graphic processing.

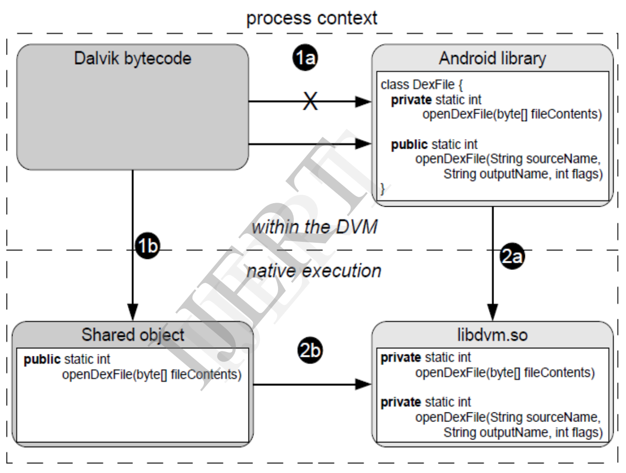

process context
Dalvik bytecode
1a
Android library
class DexFile {
private static int openDexFile(byte[] fileContents)
public static int openDexFile(String sourceName, String outputName, int flags)
}
within the DVM
1b
2a
native execution
Shared object
public static int openDexFile(byte[] fileContents)
2b
libdvm.so
private static int openDexFile(byte[] fileContents)
private static int openDexFile(String sourceName, String outputName, int flags)

Figure 8. *Possible control flows to load a further dex file into a running process*.

Some other obfuscation technique[3] are mention below.

Insert Dead or Irrelevant Code
The insertion of dead code or junk code confuses the attacker. You insert code that will never be executed and/or will never contribute to the functionality of the program."This code can include extra methods or simply a few lines of irrelevant code" [6]. It is important to note here that this dead code is to confuse the decompiler and the attacker.

Extend Loop Condition
Complicating the loop conditions introduces obfuscation in the code. This can be done by extending the loop condition with a second or third condition that doesn't do anything [6]. For example, in the following example we have a simple if condition.

Before:

```
int x = 1;
if (x > 200)
{
   …
   x ++;
   // call function
   abc(x)
        }
```

After:

```
int x = 1;
while (x> 200 ||
   x%200==0)
{
   …
   x ++;
   // call function
   abc(x)
 }
```

Add Redundant Operands

Adding some insignificant terms to the code, in the basic calculations confuses the reverse engineer. For example, let"s assume that there is an integer variable, 'p' that stores the product of two integer variables – 'a' and 'b'. The code below shows we can make the calculations look complex to the attacker [6].

Before:

```
public int sum{
     int a = 5;
     int b = 7;
     int p;

     p = a * b;

System.out.println("
Product =" + p);
}
```

After:

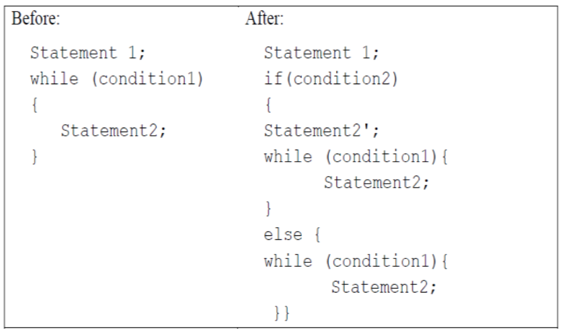
```
public int sum{
     int a = 5, b = 7;
     double i = 0.0005;
     double j = 0.0007;
     double p;

     p = (a * b) + (i*j);
     System.out.println("
Product =" + (int) p);
}
```

Clone Methods

It is important for a reverse engineer to understand the purpose of a function and it is equally important to understand the different conditions under which the function is called [10]. We can create clones of a function and make calls to these functions under identical circumstances. We can call the function depending on any external factor, which appears to be a deciding factor but is actually not. One good example would be to call a different function based on the day of the week[6].

Reducible to Nonreducible"

The Holy Grail of obfuscation is to create obfuscated code that cannot be converted back into its original format" [6]. We can devise some transformations that make the code nonreducible to its original form. For example, the Java bytecode has goto instruction while no equivalent statement exists in the Jav language. So, the flow graphs produced from Java programs are always reducible, while those from Java bytecode may express non-reducible flow graphs. Expressing non-reducible flow graphs is inconvenient in Java due to unavailability of goto statements, so we need to do some transformation for converting the reducible flow graph into a non-reducible one. We can achieve this by converting a structured loop into a loop with multiple headers[10]. For example, see the code below.



Before:

```
Statement 1;
while (condition1)
{
   Statement2;
}
```

After:

```
Statement 1;
if(condition2)
{
Statement2';
while (condition1){
      Statement2;
}
else {
while (condition1){
       Statement2;
 }}
```

In this example, we had a simple while condition. We split the statement to make it appear more complicated than it actually is.

3. Proposed Work

The problems that a developer is facing today because of reverse engineering are listed below.
(a) *Remove Restriction* - A software developer generally release their products as evaluation version with some limitation. By Software Reverse Engineering (SRE) one can remove this restriction and can use it as full version. For ex- Microsoft releases their operating system for evaluation purpose/trial purpose. But by reverse engineering this products one can create patches to remove trial restriction.
(b) *Cheat code of Games* - Cheat code for games can be made by using SRE.
(c) *Crack & Patch of software/games* - By SRE one can make crack & patch of a software/games. However it's not an easy job but expert can do this.
(d) *Cloning a software* - One can make exact replica of existing software with the help of SRE as reverse engineering is not an illegal process as per US & Europeans laws.
In this paper we are adding few more logic on android source code to make it more complex to understand. This will make a software more secure against reverse engineering.

We are using try-catch blocks to change the flow of application at runtime. Normally the try-catch block are used to deal with exception. Try block contains the business logic and if that logic throws any exception then it can be handled in catch block. The catch block contains code to deal with exception. As per approach of this paper all the business logic will be written in catch block and the try block will have unnecessary dead code, meaningless loop, and senseless if-else condition. Also there will be one line which will always cause exception at runtime. So in runtime when dalvik virtual machine will execute the code of try block then that particular line will throw error and the flow will go to catch block and here in catch block our business logic is already written so it will get executed. While on viewing the code by reverse engineering one will focus to understand the try block and as the try has lots of senseless, meaningless code this will definitely irritate the reverse engineer.

```
try
{
Public Class A()
{
        //        UselessCode();
}
Public Class B()
{
        //        AnotherUselessCode();
}
```

```
AobjA;
B objB=new B();
objA=(A)objB;              ---- This will lead to NullReferenceException and rest of the code
FewMoreLinesOfUseLessCode();  --- Will not execute at runtime. It's just to confuse
-------;
-------;
Catch(exception ex)
{
        ActualBusinessLogic();
}
```

Figure 9.*Example of proposed try-catch block approach*

To make things more complex to understand we can use the existing prevention technique like code obfuscation (Renaming Class, Method, variable name to meaningless letter/word). Also few more confusing existing logic like if-else, for loop can be put in the catch block to secure our actual business logic.

4. Future Scope

To achieve 100% anti reverse engineering technique one will have to go through the complete logic how a decompiler identifies the **dex** file from the **apk** package and how it interprets the code of **dex** file. If the actual working of decompiler is recognized, with the implementation of suitable approach on the **dex** file, decompilation of **dex** file can be controlled. Since decompiler cannot identify **dex** file from **apk** package. Hence the application is secured from the reverse engineering.

5. Conclusion

Android is basically built on top of the Java and Linux platform. Both are open source so obviously reversing the app is easy compare to other platform based application. By applying proposed approach we are making the code & flow more confusing for a reverse engineer. This will make the reverse code even more harder to understand as we are changing the dynamic view of the code as well as static view together to confuse the reverse engineer. Only experts reverse engineer can break this complex structure, so we can at least prevent our application to be reverse engineered from the normal reverse engineer.
Still if application is dealing with highly secure system like banking system, reservation or some payment gateway then it would be better to put all the business logic on the server side. This will be best practice to prevent source code from the reversing.

Acknowledgment
We would like to thank Mr Ishan Malhotra and Miss Ashima Sahu for their valuable support during research work.

References

[1] Patrick Schulz "Code Protection in Android " 2012-Schulz-Code_Protection_in_Android.pdf .
[2] Manjunath,Vibha " Reverse Engineering Of Malware On Android"(2011).[https://docs.google.com/viewer?a=v&q=cache:rm7KrxkEYgsJ:www.sans.org/reading_room/whitepapers/pda/reverse-engineering-malware-android_33769+research].
[3] Kundu, Deepti, "JShield: A Java Anti-Reversing Tool" (2011).Master's Projects. Paper 161.
[4] Android Open Source Project. Android sources. Visited: May,2012. [Online]. Available: http://source.android.com
[5] Venkatesan, Ashwini, "Code Obfuscation and Virus Detection" (2008).Master's Projects.
[6] Nolan, G. (2004). Decompiling Java. Chapter 4 – Protecting Your Source: Strategies for Defeating Decompilers, pages 79 – 210. New York, USA: Springer-Verlag New York.
[7] "Code Obfuscation against Static and Dynamic Reverse Engineering" Sebastian Schrittwieser and Stefan Katzenbeisser.
[8] android-apktool project [Online]. Available at: http://code.google.com/p/android-apktool/ (July 2011)

[9] smali project [Online]. Available at: http://code.google.com/p/smali/ (July 2011).
[10] Collberg, C., Low, D., & Thomborson C. (1997). A Taxonomy of Obfuscating Transformations. *Technical Report.* Department of Computer Science, University
of Auckland, New Zealand. Retrieved October 21, 2010 from
http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson
Low97a/A4.pdf
[11] Pro Guard [Online]. Available at: http://proguard.sourceforge.net/
[12] Gartner. Worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth. Visited: May, 2012. [Online].Available:http:www.gartner.com/it/page.jsp?id=1924314

Sudipta Ghosh is currently pursuing M.Tech .from Dr C V Raman Institute of Science & Technology, Bilaspur. She received her B.E.(IT) from Guru Ghasidas Central University, Bilaspur, Chhattisgarh, India. Her interest area includes Software Engineering, Android Application and Machine Learning.

S.R. Tandan is Currently Assistant Professor in the department of CSE, and Pursuing Ph.D from Dr. C.V. Raman University, Bilaspur, Chhattisgarh, India. He received his M.Tech.(CS) form BITs Mesra and BE(CSE) from NIT, Raipur, His interest area includes Soft Computing, Information Retrieval System and Mobile Robot Navigation, and Cyber Crime.

Kamlesh Lahre is Currently Assistant Professor in the department of CSE. He received his M.Tech(CSE) form NIT,Raipur and BE(IT) from GEC, Bilaspur, His interest area includes Application of Soft Computing.