Code Obfuscation of Android Applications

COMPSCI-702-Group 6
The University of Auckland

May 12, 2016

1 Summary

The big adoption of new market trends such as Bring Your Own Device (BYOD), Mobility and others led to a huge increase in the number of mobile applications. Accordingly, protecting these applications becomes of high interest. A large number of protection techniques have been developed and code obfuscation is one of these techniques. It is a mechanism for hiding the original algorithm, data structures, the logic of the code, or to harden and protect the code from the unauthorized reverse engineering process [1]. This paper will discus code obfuscation as a method of protection along with a number of its techniques, focus on code obfuscation implementation on Android devices as a deployment platform through the development of an Android application and obfuscation tool that will utilize a number of these techniques. Additionally we will discuss the evaluation of the developed tool from different aspects, the reverse engineering attempts of other groups' obfuscated applications along with future areas of improvements.

2 Introduction

Context

With the huge development of applications and Internet technologies such as Cloud, Virtualization, Mobility and others along with the large adoption of smartphones and mobile devices, organizations and IT industries are facing more security challenges to protect their data, intellectual property as well as data processing within the application from software piracy or injection of malicious codes [1]. Application software protection is a continuous process where different methods, approaches and techniques are developed, tested and implemented to defeat attackers and achieve strong protection levels. One of these methods is code obfuscation, which is our focus in this paper.

Problem

Reverse Engineering is the process of reconstructing the original application source code out of the distributed binary form by gathering as much information as possible in order to understand the code functionalities. Obfuscation cannot prevent reverse engineering but can make it harder and more time consuming [2].

There are three main techniques for reverse engineering of software [3]: 1- Observation of data exchange. 2- Disassembly: The process of reading the binary code of a program by a specific software called disassembler. 3- Decompiling: Recreating the source code of a program from the binary code or bytecode.

Reverse engineering of software has a serious impact and can cause a number of issues including: 1- Malicious software: Vulnerabilities in operating systems and other software programs can be exploited. 2- Cryptographic algorithms can be exposed or broken. In addition, the encryption and decryption keys can be obtained, extracted or generated.

State-Of-The-Art

Researchers and software industries are continuously trying their best to apply newer and better techniques to protect their intellectual property. There are a number of techniques and approaches

implemented such as code replacement/update and code tampering detection. Additionally, there are common methods used to avoid or challenge the detection engines such as encryption, protection by server-side, hardware-based security solutions, different signed native codes, watermarking and software aging. Code obfuscation is another powerful approach that can be used to protect the applications and it can also be combined with other methods such as the ones mentioned previously to achieve better strength [1].

Solution

In this paper, we will focus on code obfuscation as the main method to protect Android applications and programs. Code Obfuscation is the process of transforming the source code or intermediate code of an application or program to make it more difficult to be decompiled and analyzed [3]. Obfuscation hides program semantics through choosing semantically equivalent but complex and ambiguous representations without changing the behavior of this application [2].

Figure 1 highlights the build process of an Android application.

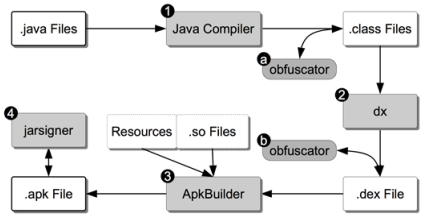

Figure 1: Android Application Build Process [2]

Obfuscation can be applied using automatic tools called obfuscators at different stages throughout the build process such as step "a" that is applied on the ".class" files and step "b" where obfuscation is applied on Dalvik bytecode. Additionally, there are other techniques to apply code obfuscation as we will discuss later.

In this paper, we are developing an obfuscation tool that utilizes a number of open-source tools and enhancing them with other techniques to increase level of complexity. We are mainly utilizing open

source powerful obfuscation tool called “Ob” using its String Encryption (StringEncryptor.class) technique and used hashlib library to obfuscate all methods and variable names using SHA256 algorithm whilst maintaining the legality of the names in Java as well as introducing dummy classes and methods into the code.

Novelty

The sole purpose of injecting bad code in the source code is to defy popular decompiling tools. Most of the times, a simple combination of known exploits is enough to cause certain tools to produce an output error. Decompilers and disassemblers performing static analysis are usually susceptible to such exploits.

To thwart decompilers, an advantage is taken from the discrepancy between what is representable as legitimate Java code and its translation into bytecode or Dalvik bytecode. Java programming language does not implement a goto statement, yet when a loop or switch structures are translated into bytecode, this is done with a goto Dalvik instruction. Thus by working directly on bytecode, it is possible to inject correct sequences composed of goto statements which either cannot be processed by the decompilers or do not translate back to correct Java source code. In our method of obfuscation, we have injected bad code to confuse and/or break popular decompilers.

3 Related Work

There is a large number of commercial obfuscation tools available in the market today. These tools implement different obfuscation techniques. In this section, we will highlight a couple of these tools [2]:

- ProGuard is a Java obfuscation open source tool that is integrated in the Android SDK. This tool utilizes identifier obfuscation technique that includes classes, methods, and fields. ProGuard can also be used to remove dead code as well as unused classes.

- Allatori is a commercial product that uses the same identifier obfuscation technique as ProGuard as well as methods to modify the program code to make algorithms less readable. Additionally, strings such as messages and names are obfuscated and decoded at runtime.

In addition to the commercial tools available, there have been a number of papers proposing new obfuscation techniques. Jan M. Memon [4] proposed two Java bytecode obfuscation techniques; Unletting the completion of statement and Removing variable and method name. These techniques introduce syntax and semantic errors in the generated source code, which prevents automatic software analysis tools and de-compilers from producing correct source code.

Additionally, Marious Popa [3] discussed a number of obfuscation techniques in terms of functionality and capability:

- Name obfuscation. It is the process to replace the identifiers with meaningless strings as new identifiers.

- Data obfuscation: This technique changes the ways in which data is stored in the memory.

- Code flow obfuscation: This involves changing the control flow in an application in a way to make it more difficult to be analyzed but without affecting the initial result.

- Intermediate code optimization: This affects the size of the intermediate code. Optimization techniques include: 1- Removing the unused methods, fields and strings. 2- Evaluation of constant expressions. 3- Assignment of static and final attributes.

- Debug information obfuscation: Removing the debug information that can help discovering the original source code of the program.

- Watermarking: Embedding information in the software application that prevent unauthorized software disclosure.

- Source code obfuscation: Hiding the source code when it is disclosed to a third parties by renaming the program identifiers and removing the comments.

Chandan and D.lalitha [1] proposed obfuscation techniques that are implemented in assembly level code such as "combining binary instructions with Assembly code" and "Use of decimal numbers in between Assembly code instruction".

Our proposed idea is taking an advantage of the discrepancy between what is representable as legitimate Java code and its translation into bytecode or Dalvik bytecode. This is done by working directly on bytecode and injecting bad code to confuse and/or break popular compilers.

4 Proposed Idea

In this project, we have developed an Android application as well as an obfuscation tool. The idea of the Android application is to use an algorithm that will match Medicine students and Hospitals based on specific criteria. A user can input a medicine student and his/her hospital preferences as well as hospitals and preferred choice of medicine students as another approach. This information will be stored in a SQLite database and then the algorithm will generate a perfect match between the medicine students and hospitals.

The application is developed as an Android studio project, which includes a Java library as compile dependency and the algorithm is located in that java library. The application is composed of a User Interface (UI) and a service that implements the library.

The other part of our project is developing the obfuscation tool. The approach of developing our own obfuscation technique and tool would be extremely difficult in the given time frame. Therefore we

decided to look for some open-source obfuscation tools and enhance them to suit our Android application.

The process has gone through a number of stages:

Stage 1: We started working on an obfuscation tool called "JMOT" that was based upon Apache BCEL (Byte Code Engineering Library). After going through the source code, we worked on enhancing the strength of obfuscation and making it difficult for the decompiler to decompile the obfuscated code. We injected some trailing instructions after an xreturn (ireturn, areturn, return) instruction to confuse the JD-GUI decompiling tool and break the functionality to decompile a JAR/Class file into a Java file. We also checked the source code for a signal to some unreachable branch code so that we could inject useless bytecode into this branch to make it difficult for the decompilers, without altering the stack height.

Stage 2: In this stage, we were able to discover another open source tool; "Ob" that is also based on BCEL. This was better and more powerful than "JMOT" with more complex obfuscation, mainly on four aspects:

- String Encryption (StringEncryptor.class)
- Unconditional Branch Transformation (UnconditionalBranchTransformation.class)
- Class Renaming (ClassRenamer.class)
- Field Renaming (FieldRenamer.class)

We then ported our previous work onto this new tool and injected trailing instructions with "ireturn" after normal unconditional branch instructions like "goto" and "xreturn". We also increased the complexity of the useless bytecode that we injected in the unconditional branches.

After testing our obfuscation tool on the android application, we ran into some problems and the obfuscated android application was not working. The output from our obfuscation tool (with String

Encryption, Unconditional Branch Transformation, Class Renaming and Field Renaming) made changes to the source code in such a way that Android Studio would show up errors related to class names and would not compile.

Unconditional Branch Transformation, Class Renaming and Field Renaming caused the main issues and since removing these techniques will reduce the complexity, we implemented source code obfuscation in lieu of these techniques to maintain high level of complexity.

Stage 3: We used hashlib library to obfuscate all methods and variable names using SHA256 algorithm whilst maintaining the legality of the names in Java. We then introduced dummy classes and methods into the code to serve as distractions for people trying to reverse engineer our code. We also refactored the code so that most of our method calls called another class or interface.

```
—code starts—
import hashlib
import sys

names = sys.argv

/*for name in names */

temp = hashlib.sha256(name.encode("UTF-8")).hexdigest()
            if list(temp)[0]>='0' and list(temp)[0]<='9':
                        temp+='a'
                        temp = temp[::-1]
            print(name,temp)
            print()
—code ends—
```

Figure 2: Hashlib Code

5 Evaluation

In our implementation of Gale-Shapley algorithm [5], obfuscation doesn't affect the efficiency of our application. The algorithm is stored in a java library and used as a compiled dependency whereas the application part contains only the GUI and a service which implements the library. There was no noticeable difference in perfor-

mance overhead (execution time or memory heap) or storage overhead between our application's normal version vs. its obfuscated one.

There are three different kinds of app startups: 1- First start: It occurs when a user runs the application for the first time after a fresh installation. During the first start, Android or the application does some extra work, such as initializing SQLite. 2- Cold start: It occurs when a user runs the application after he/she hasn't used the application for a while. If an application is not used for a while, Android typically removes the application from the cache to save memory. Cold start is generally the most typical case, and affects UX the most. 3- Warm start: It occurs when a user switches away from application then switches back. The application is still in Android's cache, so warm start is typically fast.

There were no memory leaks found in both our versions of the application.

We used [6] to run the tests. The screenshot with file name app-release.apk is non-obfuscated while file name with COMPSCI-702-2016-G6.apk is obfuscated app.

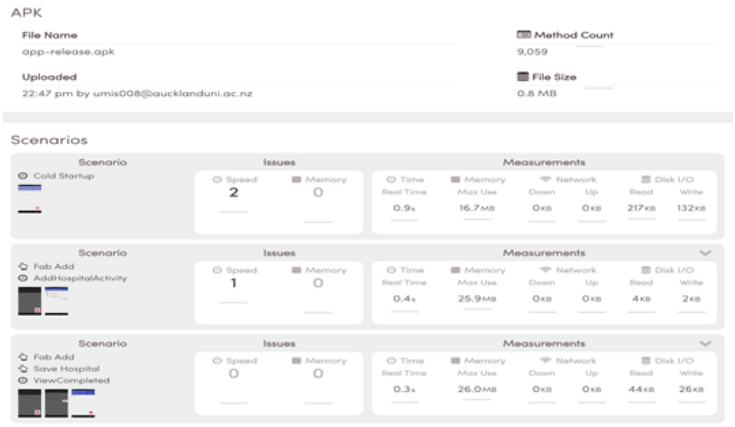

Figure 3: Non-Obfuscated Application Measurement 1

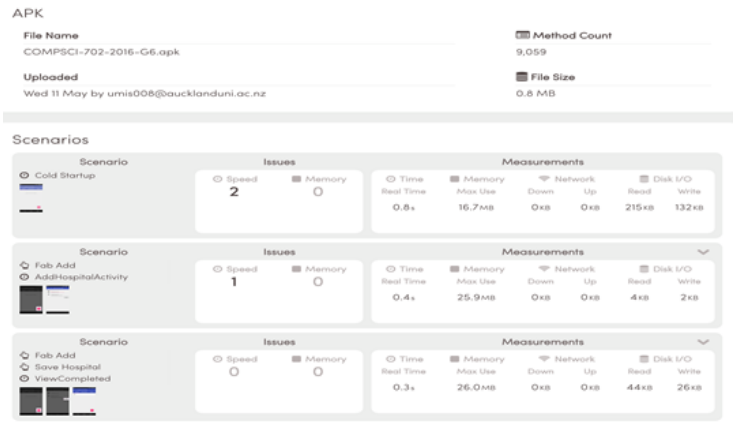

Figure 4: Obfuscated Application Measurement 1

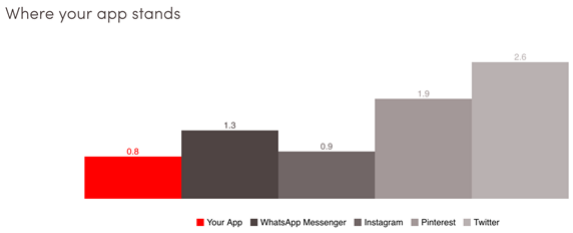

Figure 5: Non-Obfuscated Application Measurement 2

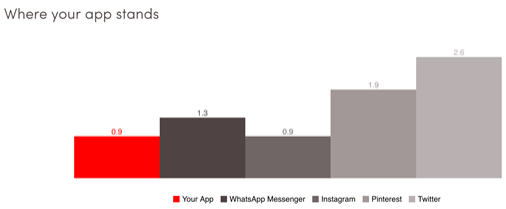

Figure 6: Obfuscated Application Measurement 2

Status of reverse engineering

Group 1

This application uses basic steganography techniques to hide a message in an image. The application's MainActivity allows the user to select an image from the device's SD card and set a message they wish to hide in that image. The EncoderService and RandomPointGenerator classes are then used to hide the message in certain pixels of the image. When the user wishes to retrieve the message from the image the DeencoderService and RandomPointGenerator classes find the locations in the image where the information is stored and retrieve it.

Group 1's primary obfuscation technique was to obfuscate class names and field variables with a binary number counting system where 1 and 0 were replaced with capital I and l. The purpose of this was to try to obscure their meanings and make it harder for humans to differentiate between variables.

Group 1's secondary obfuscation technique was to break up their programme logic into single calls to static methods in other classes. Those classes would in turn call static methods in yet more classes meaning that the app was full of "dummy" classes. Most of these "dummy" classes do nothing more than call other classes as shown in Figure 7 below.

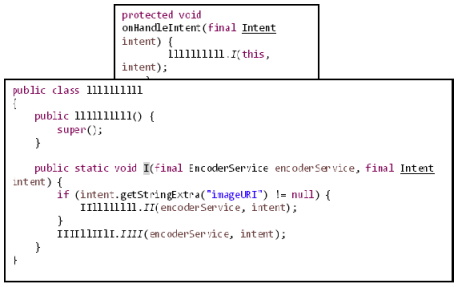

Figure 7: An Example of the static method calls used to break up the programme logic

Group 2

Group 2 used a very particular approach to prevent the original source code being obtained unauthorized. Instead of using code obfuscation, they use an existing tool called DexUnShell to re-inforce the original apk file. They first wrote the functional code in the normal way, and then compiled the source code to an apk file. Then they wrote the proxy application (works as a shell and the decryption tool) from the template provided together by that tool. The proxy application is then compiled and get the dex file out.

To reassemble the proxy and actual application, DexUnShell is then applied. The tool uses a byte operation and a key "ivy" to encrypt the apk byte and put the actual apk's encrypted byte behind the proxy dex byte. To make the assembled dex still correct, the tool will recalculate the dex file header, checksum and SHA1. The ouput dex is then put into the apk and replace the dex with in it.

At the runtime, the proxy application class and re-invoke class will recover the actual bytecode from the dex file and regenerate the actual application dynamically. Those who attempt to reverse engineer the apk will only get the proxy application, which does not make any sense to what the application does.

To get the actual application source code, there is two ways to do that. First one is to dump the android memory when the proxy is regenerating the actual apk. The second way will be compile the reverse engineered proxy application, get the dex file, and compare it with the dex we get from the apk. The byte difference will be the encrypted bye of the original apk. And since the proxy application class have the decryption method, we can write a tool to do the decryption, and then do a de-compile on the original apk, which have no obfuscation. Overall, this is a very effective protection to the source code, but not an obfuscation. To make the protection work better, they can add some code to the proxy application so that people cannot tell that it is not a proxy.

Group 3

They used String Obfuscation and anonymous class as their obfuscation techniques. Packages used in this application: 1- Package

(a) is where the string obfuscation is decrypted. 2- Package (Android) is the library for Android framework. 3- Package (com.google.android.gms) is where google map service located, which provide location-based function for the application.

They used base64 and a custom algorithm to encrypt the string and stored the encrypted string in a messages.properties file with a key-value map. Then they used a key within the application and called decrypted method in package a to get plain text. We created a tool which use the code found in their decryption part with some modification, we are able to decrypt the string used in this application.

Group 4

The obfuscation techniques they used: 1- AES plus base64 encryption for all strings including SQL query. 2- Some control flow obfuscation method that messed up the original control flow with try catch and goto. This method is very successful in confusing all the decompilation tools tried where the original byte code needed to be analyzed to understand the code. 3- Class/method renaming to remove the meaning represented by the original name. 4- Inserting some middle classes to increase the difficulty of analysis.

Group 5

Group 5's obfuscation techniques were to use base64 encoding on their strings and to add branch statements which will never execute one of the branches. Along with this they also used some basic name obfuscation. The base64 strings could be de-obfuscated quite easily as can be seen in Figure 8.

Figure 8: An example of base64 decoding from the group 5 application

The misleading branch statements were not particularly difficult to de-obfuscate since the Boolean values were constant and it wasn't hard to work out what those constants are. Figure 9 shows such a branching statement, where the method "fakebranch" will always return false for the value 1086.

```
private static boolean fakeBranch(int i) {
            if ((((double) i) / 2.0d) % 2.0d != 1.0d) {
                return true;
            }
            return false;
        }

        public void onAttach(Context context) {
            super.onAttach(context);
            if (C0926A.fakeBranch(1086)) {
                C0592E.L1L1L1L1L1L1L().L1L1L1L1L1L1L1L();
            } else
            if (context instanceof C0590B) {

        this.L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1L1 =
(C0590B) context;
            } else {
                throw new RuntimeException(context.toString());
            }
        }
```

Figure 9: An example of a misleading branch in the ListActivity class

De-Obfuscation Techniques: Through the use of tools such as BytecodeViewer.2.9.8 and `http://www.javadecompilers.com/apk`; which is an online decompiler, reverse engineering this application was relatively easy. Next we used a tool, Android application x, to decode the base64 encoded strings. Then through the use of an IDE like Eclipse we worked through the program logic removing the misleading branch statements.

Group 7

Group 7's primary obfuscation technique was name obfuscation. They appear to have used an algorithm such as the SHA1 hashing algorithm to produce garbled names for methods and variables. The purpose of this was to try to obscure their meanings and make it harder for humans to differentiate between variables. The secondary obfuscation technique was some minor flow control obfuscation. This "confused" the reverse engineering tool we used and produced source code which did not compile. See Figure 10

```
...
switch (i)
   {
   default:
   case 2131492975:
   case 2131492977:
   case 2131492976:
     do
     {
       do
       {
         return;
this.mze7416cgy77np69y47cp778m8232qy8483swkjf9072g92lux21koe22
96lz74887161.setText(this.olaht97301llc9484596423b27h61n20vo2343to
2179q17h84p091ir51lgir024s7069.mkt70q60zvmf6489w627028964896so
91g14543313dju31930na43w2498py150yrz12329());
         return;
       } while
(!this.olaht97301llc9484596423b27h61n20vo2343to2179q17h84p091ir51l
gir024s7069.eqd5820wr18sfa694n94cw9517de22lj4627ug513mq6478181
92d2163xnk79817670s30());
...
```

Figure 10: An example of un-compilable code from the onClick method of PlayerActivity class

Group 8

Group 8 used a very limited source-code obfuscation techniques in their application (no JVM/DEX bytecode obfuscation technique used):

1- Adding thousands of junk classes/methods: a. Easily fixed by just removing the useless stuff with many easy tools. b. Brings very bad overhead for the application 2- Class/method rename: Basically useless since it just adds a few reading difficulties. 3- Adding an infinite loop in an unreachable branch marked with string “opening” to confuse the reader: Easily discovered by checking the running status of the application. 4- Mirroring all the classes: Two sets of exact same classes and both of them are invoked.

6 Discussion

Limitations

Our application works on matching the Gale-Shapely algorithm [5] creating a pairing between hospitals and medical students, but there are some restrictions to our application. This release of our application crashes on pressing "Find Matches" if the following certain steps are not followed:

Step 1: While entering a hospital, the user must give it a name and a preference list which contains all the students that they will create in step 2. This preference list must be delimited with the "," character. For example: Hospital name: Frank Hospital preference list: Kate,Mary,Rhea,Jill. Hospital name: Dennis Hospital preference list: Mary,Jill,Rhea,Kate. Hospital name: Mac Hospital preference list: Kate,Rhea,Jill,Mary. Hospital name: Charlie Hospital preference list: Rhea,Mary,Kate,Jill.

Step 2: In this step the user must create ALL the students that they entered into the preference lists of the hospitals in step 1. The creation of a student is the same as the creation of the hospitals. Also each student must have a name that name must appear in all the preference list of the hospitals from step 1 and all of the hospital names from step 1 must appear in each student's preference lists. For example: Student name: Rhea Student preference list: Frank,Mac,Dennis,Charlie. Student name: Mary Student preference list: Mac,Charlie,Dennis,Frank. Student name: Kate Student preference list: Dennis,Mac,Charlie,Frank. Student name: Jill Student preference list: Charlie,Dennis,Frank,Mac.

Step 3: Press the find matching button and then the application will create and display the results. The user may need to scroll to refresh the screen so they can see the results.

If these steps are not followed, the application crashes. Our application includes a method, which encrypts the names of the students and hospitals as well as the preference lists. Use of this method restricts the input to letters of the alphabet, not allowing numbers to be entered in the name or preference list fields. In hindsight we

wouldn't have added this method to our app but we included it for the sake of increasing its complexity and confusing other groups trying to de-obfuscate it. If this method were to be removed, it would increase the range of inputs that could be entered and make the application more stable.

Possible Extensions

In this release of our application, we followed the "Stable Marriage" algorithm proposed by David Gale and Lloyd S. Shapely in 1962 [5] for matching hospitals and medical students according to their preferences. However, in a real world scenario, there are way more complex situations, which are not addressed in this release. For example, if a student gains admission to their preferred school after having been placed on a waitlist, they may have already accepted admission to a less-preferred school. In this case, the student will have to study in a less-preferred college. In another case, some students may feel that listing every college as their first choice on each application will improve their chances of admission. Assuming that colleges reward applicants for listing their college as the first choice and do not share application information with each other, the applicant will occupy valuable seats/slots at a number of colleges while will attending only one. Considering this scenario, our application can be extended, in future, to include new possibilities and real world situations to provide a better match between medical students and hospitals providing each college a deferred acceptance procedure and eliminate uncertainties.

References

[1] Behera, Chandan Kumar, and D. Lalitha Bhaskari. "Different Obfuscation Techniques for Code Protection." Procedia Computer Science 70 (2015): 757-763.

[2] Schulz, Patrick. "Code protection in android." Insititute of Computer Science, Rheinische Friedrich-Wilhelms-

Universitgt Bonn, Germany 110 (2012).

[3] Popa, Marius. “Techniques of Program Code Obfuscation for Secure Software.” Journal of Mobile, Embedded and Distributed Systems 3.4 (2011): 205-219.

[4] Memon, Jan M., Asghar Mughal, and Faisal Memon. “Preventing reverse engineering threat in java using byte code obfuscation techniques.” Emerging Technologies, 2006. ICET’06. International Conference on. IEEE, 2006.

[5] David Gale and Lloyd S. Shapely “Stable Marriage” algorithm. (Online; Last Accessed May 11, 2016). [Online]. Available: `https://en.wikipedia.org/wiki/Stable_marriage_problem`

[6] Nimbledroid. (Online; Last accessed May 11, 2016). [Online]. Available: `https://nimbledroid.com`

Appendix

Group 1 Code Analysis

Through the use of tools such as BytecodeViewer.2.9.8 and jd-gui-1.4.0 reverse engineering this application to obtain the obfuscated source code was relatively easy. Next we used a Python script to assign more human readable names to variables and classes. Then through the use of an IDE like Eclipse it was a trivial but laborious task to follow the static function calls through the various classes and replace them with the original logic.

The Python script was used to convert the names to a binary number and then convert the binary number to a decimal number preceded by the $ character. Whilst this did not immediately add more meaning it did however make the code easier to read. We also used some common sense naming where it was clear what the variable represented. For example Figure 11 shows the obfuscated

field variable names of the EncoderService class on the left and on the right the de-obfuscated names.

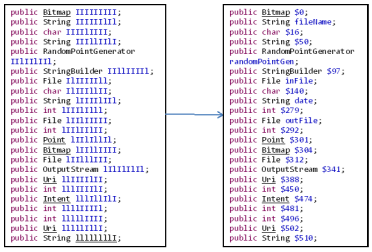

Figure 11: Example of name de-obfuscation in the EncoderService class

As shown in Figure 12 the programme logic of hideText method of the EncoderService class can be completely recovered by simply following the static method calls to their eventual conclusion. This de-obfuscation technique generalises to the rest of the application so that all of the programming logic can be recovered.

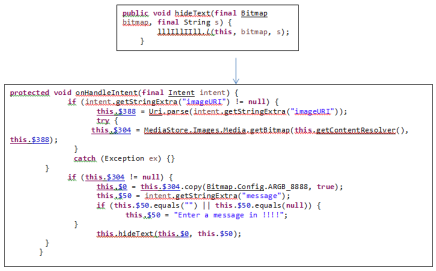

Figure 12: An example of de-obfuscating a method in the Encoder-Service class

Obfuscation Evaluation: These obfuscation techniques are not very strong. The name obfuscation technique can be mitigated by renaming the variables to make them easier to read. Once the application's programme logic has been recovered, it is simply a matter of reading through the code to understand what each variable does and

then assigning a more appropriate name. This is not usually necessary since in most cases where reverse engineering is required a general understanding of how the application works and not an exact reconstruction is good enough. The obfuscation technique of breaking the programme logic into small pieces then distributing them through other classes and linking them through static method calls can also be easily de-obfuscated. As mentioned earlier one can simply use a tool like Eclipse or any other good IDE to follow the static method calls and recover the application's programming logic. The technique of breaking up the programming logic into small pieces and linking them through static method calls appears to be quite unique. This is probably because it is ineffective and unnecessarily bloats the size of the application and code base.

Group 3 Code Analysis

The main entrance of this application is (MapActivity.class) where int a() is used to get alarm distance from setting. void a(Latlng) handles the marker on the map. (BackgroundServices.class) handles most of the business logic. In this class, this appuse pendingIntent to alarm user to get off the bus, Void a(String, d) is used to send the notification, boolean b(BackgroundServices) is used to calculate the distance to destination, void c(BackgroundServices) to start the alarm. Enum d is used to tag the start and stop of the service. (g.class) is used to generate LatLng object from Geocode. (e.class) handles the input change of the address (f.class) checks if the service is up and running

The typical routine of how this application works is like the following: show map (MapActivity.class) then get address from user input(e.class) then generate Latlng object for tracking(g.class) then start the service to start the location tracking (f.class and b.class) then wake user when approaching destination(BackgroundServices.class)

Group 4 Code Analysis

1- Control Flow obfuscation method is not a good method since the bytecode can be combined with not very accurate decompiled source code to understand it. 2- Class/method renaminig is not a sig-

nificant method since the original names have been removed permanently. 3- Inserting some middle classes is also not a good method; it just tries to write down the functionalities of every class/method.

Additionally, Group 4 has broken the rule, which limit the source code by 1000 lines.

Regarding point 2, Group 4 invoked google gms and Android support packages which made these packages to be included into the source code, but they didn't prevent these standard packages from being obfuscated. This significantly increased the lines of code from a thousand to multiple tens of thousands of lines (all the google code (around 2000 files 4M bytes) is modified and cannot be easily understand).

Group 5 Code Analysis

This application is a basic scheduling application. It allows the user to check into a "lab". Only the 702 lab is supported and the way it does this is by checking the current time with the predefined times of the 702 classes.

Obfuscation Evaluation:

This obfuscation technique was moderately difficult to de-obfuscate due to the number of distractions in the code base like the misleading branch statements. The base64 encoding was quite easy to de-obfuscate since it is a well-known encoding method used in web applications for quite some time now. Although we did not de-obfuscate all the variable and class names given more time to spend reading and understanding the code we would have been able to (almost exactly) recover the original source code.

Group 7 Code Analysis

This application finds .mp3 and .wav files and puts them in a list. This list is controlled by the MainActivity. The user can then choose a section from this list as a playlist. The playlist is controlled by the PlayListActivity and PlayList classes. The user can then choose to clear or play the list. If they choose to play the list then the media player is started. This is controlled by the PlayerActivity and the Player classes, which implemented all the functionality of the media player such as the play/pause, next and previous buttons.

Through the use of tools such as BytecodeViewer.2.9.8 and jd-gui-1.4.0 reverse engineering this app to obtain the obfuscated source code was relatively easy. Then through the use of an IDE like Eclipse renaming and editing the flow controls was quite easy. Figure 13 shows how the un-compilable code from Figure 10 has been changed so that it is now compilable and the names are human readable. The compilable code was obtained by using a few different reverse engineering tools and comparing the results of each. The names were assigned by reading through the code and picking names that made general sense.

```
...
switch (i) {
            default:
            case R.id.btPlay:

        this.playPauseButton.setText(this.player.pauseOrUnpause());
                    return;
            case R.id.btFF:
                    if (this.player.fastward5Seconds()) {
        setAlbumArt(this.player.setSongAndGetItsURI(this.player.getSong()
));

        this.songTitle.setText(this.player.getSong().getName().replace(".m
p3", "").replace(".wav", ""));

        this.theSeekBar.setMax(this.player.getDuration());
                            this.playPauseButton.setText("||");
                    }
                    return;
            case R.id.btRw:
                    if (this.player.rewind5Seconds()) {

        setAlbumArt(this.player.setSongAndGetItsURI(this.player.getSong()
));

        this.songTitle.setText(this.player.getSong().getName().replace(".m
p3", "").replace(".wav", ""));

        this.theSeekBar.setMax(this.player.getDuration());
                            this.playPauseButton.setText("||");
                    }
                    return;
            ...
```

Figure 13: Compatible code obtained through reverse engineering

Obfuscation Evaluation: These obfuscation techniques were not particularly strong. The name obfuscation could be overcome quite easily once the application's programme logic had been recovered. This is done by reading through the logic, understanding what the variable, method or class does and then giving it an appropriate name.

The obfuscation technique used to change the control flow was not effective against most of the reverse engineering tools we tried, such as some of the decompilers built into BytecodeViewer.2.9.8. This suggests that the technique is either ineffective or was done unintentionally and it just so happens that the decompiler JD-GUI,

and some others, were “confused” by this switch statement.

The obfuscation technique used to change the control flow seemed to be the most unique part of this obfuscation since we did not observe this behaviour in any other groups’ applications. However we were able to decompile the code more effectively with other tools and recover the application’s programme logic. We could then read through it and de-obfuscate the variable names relatively easily. Therefore these were not very effective obfuscation techniques.

Group 8 Code Analysis

The methods used to crack it:

1- Removing the useless stuff: There are many many way to remove the thousands of useless classes, methods, for example: python, ruby, even C.

2- The class/method rename: No way to fix since the original messages has been removed, but can change the long name into shorter ones to fasilitate analysis, this can be done by source analysing tools or even manually replace.

3- They mirrored all the classes, meaning that there are two exact same classes and both of them are invoked, so the whole footprint are doubled with no new functionality.

4- Bug in the APP - The obfuscated string showed in the application: It seems like that the obfuscation tools mistakenly replaced some strings that will be used in the activities while doing the renaming. The obfuscated string cannot be properly handled by the application itself, it showed during running.

Figure 14: Group 8 Application Bug