**Group 1 Analysis:**

**The App Enigma:**

This app uses basic steganography techniques to hide a message in an image. The apps 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. Then 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.

**Obfuscation Techniques:**

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. See Figure 1:

**protected** **void** onHandleIntent(**final** Intent intent) {

llllllllll.*I*(**this**, intent);

}

**public** **class** llllllllll

{

**public** llllllllll() {

**super**();

}

**public** **static** **void** I(**final** EncoderService encoderService, **final** Intent intent) {

**if** (intent.getStringExtra("imageURI") != **null**) {

IIllllllll.*II*(encoderService, intent);

}

IIIIllIIlI.*IIII*(encoderService, intent);

}

}

Figure 1. An Example of the static method calls used to break up the programme logic.

**De-Obfuscation Techniques:**

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. Next we used a Python script (see file decoder.py) 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 2 shows the obfuscated field variable names of the EncoderService class on the left and on the right the de-obfuscated names.

**public** Bitmap IIIIIIIII;

**public** String IIIIIIlIl;

**public** **char** IIIIlIIII;

**public** String IIIllIIlI;

**public** RandomPointGenerator IIlIIlIIl;

**public** StringBuilder IIllIIIIl;

**public** File IlIIIIIll;

**public** **char** IlIIIllII;

**public** String lIIIIlIIl;

**public** **int** lIIIlIlll;

**public** File lIIlIIIII;

**public** **int** lIIlIIlII;

**public** Point lIIlIllIl;

**public** Bitmap lIIllIIII;

**public** File lIIlllIII;

**public** OutputStream lIlIlIlIl;

**public** Uri llIIIIlII;

**public** **int** lllIIIIlI;

**public** Intent lllIllIlI;

**public** **int** llllIIIIl;

**public** **int** lllllIIII;

**public** Uri lllllIllI;

**public** String llllllllI;

**public** Bitmap $0;

**public** String fileName;

**public** **char** $16;

**public** String $50;

**public** RandomPointGenerator randomPointGen;

**public** StringBuilder $97;

**public** File inFile;

**public** **char** $140;

**public** String date;

**public** **int** $279;

**public** File outFile;

**public** **int** $292;

**public** Point $301;

**public** Bitmap $304;

**public** File $312;

**public** OutputStream $341;

**public** Uri $388;

**public** **int** $450;

**public** Intent $474;

**public** **int** $481;

**public** **int** $496;

**public** Uri $502;

**public** String $510;

Figure 2. Example of name de-obfuscation in the EncoderService class.

As shown in Figure 3 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 app so that all of the programming logic can be recovered.

Figure 3. An example of de-obfuscating a method in the EncoderService class.

**public** **void** hideText(**final** Bitmap bitmap, **final** String s) {

lllIllIIll.*l*(**this**, bitmap, s);

}

**protected** **void** onHandleIntent(**final** Intent intent) {

**if** (intent.getStringExtra("imageURI") != **null**) {

**this**.$388 = Uri.parse(intent.getStringExtra("imageURI"));

**try** {

**this**.$304 = MediaStore.Images.Media.getBitmap(**this**.getContentResolver(), **this**.$388);

}

**catch** (Exception ex) {}

}

**if** (**this**.$304 != **null**) {

**this**.$0 = **this**.$304.copy(Bitmap.Config.ARGB\_8888, **true**);

**this**.$50 = intent.getStringExtra("message");

**if** (**this**.$50.equals("") || **this**.$50.equals(**null**)) {

**this**.$50 = "Enter a message in !!!!";

}

**this**.hideText(**this**.$0, **this**.$50);

}

}

**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 app’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 app 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 app’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 app and code base.