Group 4

Aditya Krishnan, Lucas Betts,
Mohammad Ladha, Zain Khan

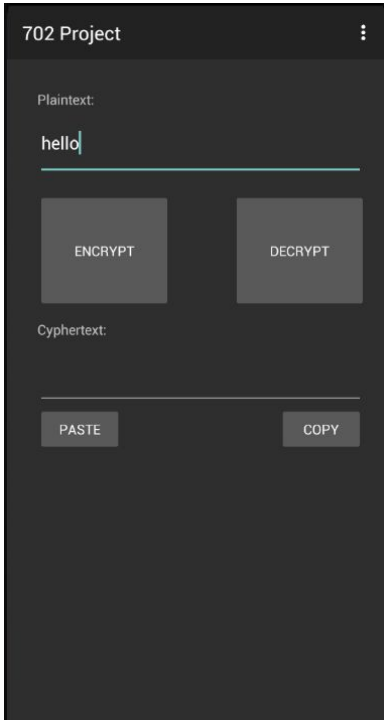
Overview

Our sample application is a simple string encryption-decryption program.

It takes a users string and can encrypt it, or decrypt an already encrypted string.

As developers, we do not want the algorithm employed to be publicly known.

In an application such as this, the underlying program may employ algorithms that are valuable intellectual property.



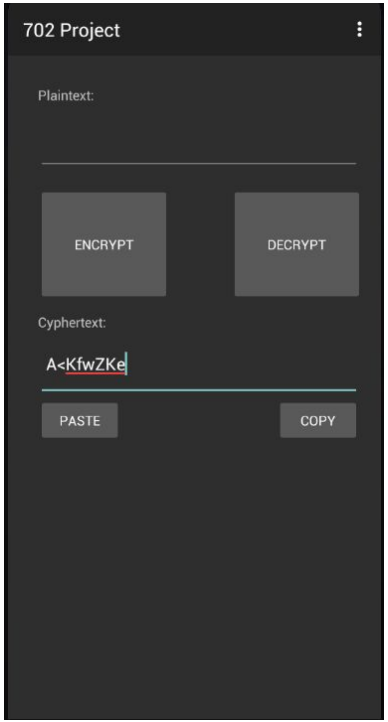
Overview

Attackers may reverse engineer an application like this in order to develop attacks against the cryptographic algorithm employed.

The goal of this project was to prevent this from happening.

By utilising a combination of control-flow obfuscation, string obfuscation and variable obfuscation, we aimed to hide the true program logic.

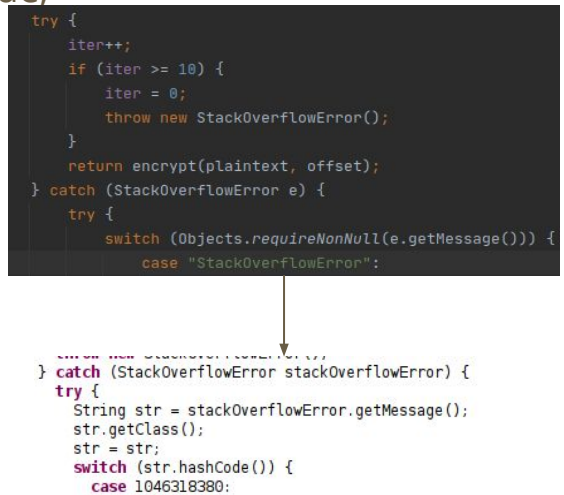
Although the original aim was to develop a tool to do this obfuscation for us, due to time constraints manual obfuscation techniques were used.



Technique #1 - Control Flow Obfuscation

To make it difficult for reverse engineers or attackers to analyze our code, we utilised dummy code.

- Dummy code is code that provides no functionality
- We used Try blocks, Switch statements and Try-with-Resources.
- Occasionally changing if a Try succeeded or failed
- Used runtime-only parameters as switches
- Try-with-Resources is not handled well by older decompilers.
- Try's and Switches are also mishandled by many, and decompiled into nested If statements.



Technique #2 - String Encryption

To hide the function of our user-interface components, we encrypted our strings and used a runtime-only decryption algorithm.

Instead of using an off-the-shelf encoding/encryption algorithm with a fixed key, we generate the key based on the last key (rolling keys/CBC).

A simple algorithm relying on a fresh sub-alphabet for each 12-character block of text.

```
A B C D E F G H ... Z a b c d e f g h ... z 1 2 3 ... 9 0 : , .
V V V V V V V V     V V V V V V V V V V     V V V V     V V V V V
L A y .   , R t     z Q I 4 e c 3 : Y     4 J h 8     K r o L a
```

In this example, “Chef1” -> “yYc3J”

Benefit of this is static analysis will not reveal the code.
Downside is difficult to generate strings, and changing any one string changes all subsequent strings.

Technique #3 - Variable Obfuscation

In a naive attempt to hide the entry point of our program, we named our main activity: “android.appcompatibility.compressionalgorithm.appcompat”

We also utilised the existing compiler options as follows:

- android.enableR8.fullMode=true
- minifyEnabled true
- shrinkResources true

These served to automate the hiding of variables in our program.

Evaluation - Strength of the Technique

- Manual obfuscation will be a problem when dealing with larger program sizes (>1000 lines)
- Control flow obfuscation using Switch statements is promising
- String encryption is useful and novel but will only be effective once

Evaluation - Performance Overhead

- Possibly Significant performance overhead
  - Will only be identified as the scale of app increases
- Compile time increased
- Resource cost increased
- Memory usage could be higher as well

Evaluation - Storage Overhead

- Effect of control flow obfuscation
  - the application will increase in a rate that is faster relative to the source code. (not exponential) due to each block of code having multiple switch statements
- Effect of data obfuscation technique
  - no significant increase in size as it’s just a one to one mapping.
- Overall Storage overhead is significant and will not scale with larger apps

Existing tools

Obfuscapk- A blackbox technique which allows users to obfuscate compiled android apps without the need of the sourc code. Supporting advanced features like

- String encryption
- Native libraries encryption
- Randomising the manifest

Pros:

Does exactly what we did and is automated hence saves time and effort

Dynamic obfuscation so each iteration of the app being compiled is different

Cons:

Easy to hide malware, acts like a double edged sword

State of the art tools and techniques

There are other tools like ADAM and AAMO which automatically obfuscate applications in a similar manner but AAMO's latest release is not stable and both of them ADAM has not been updated since it was released therefore it does not work with many newer obfuscation techniques.

Other techniques:

1. Arithmetic Branch: aims at adding some useless and semantic-preserving instructions to the code
2. Methods Overload: It exploits the overloading feature of the Java programming language to assign the same name to different methods but using different arguments.
3. Debug Removal: This technique just removes debug meta-data.The removal of debug information, such as line numbers, types,or method names, reduces the amount of useful information for the reverse engineering process.

Aonzo, S., Georgiu, G. C., Verderame, L., & Merlo, A. (2020). Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX*, *11*, 100403. https://doi.org/10.1016/j.softx.2020.100403

Limitations

- Manual obfuscation
  - It was difficult to get a tool to help with the obfuscation (e.g. javalang in Python)
  - Significant production-time overhead
  - Time constraints within the group
- String obfuscation
  - One slight change in one ciphertext changes all subsequent ciphertexts
  - The process is time consuming and cannot be parallelised

Limitations

- Scaling
  - The obfuscation techniques would not scale well with greater lines of code
  - This is due to potential for repetition of techniques - easier to trace and crack
  - High storage overhead - one block may need multiple blocks for control flow obfuscation
- Debugging
  - Stack trace outputs not useful due to the hiding of variable and class names
  - Any updates and fixing of bugs have to be done in the pre-obfuscated app
  - Additional time overhead

Group 1

Chinese variable names were present and their English translations were meaningless which indicated they used their own variable obfuscation techniques. Base64 encryption was used for the strings that were to be displayed on the app and we were able to determine which particular strings were to be displayed. Function calls in some areas were duplicated and multiple instances of the MainActivity were created, indicating control flow obfuscation was used.

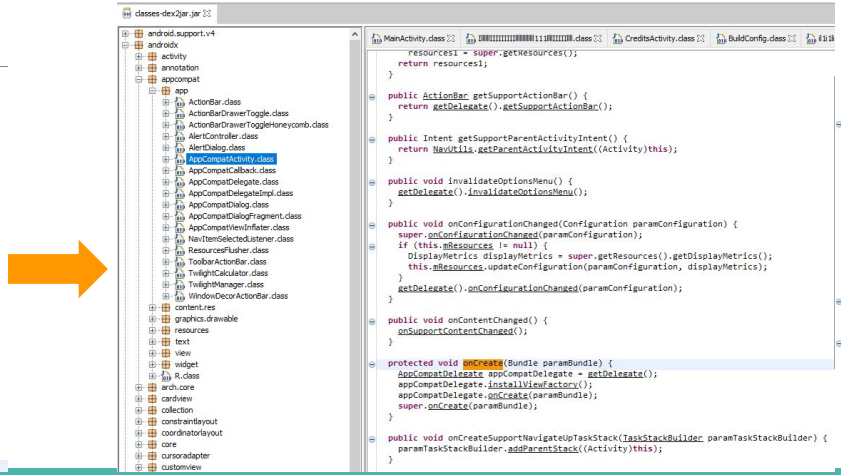
```
/* renamed from: 蠶嚳畫纙鬚籲鬚  reason: contains not printable characters
public final String m1245(String str) {
    return new String(Base64.decode(str, 0));
}
```

```
String string = C0570.m2234((Context) this).getString(Base64Decode("cHJlZmVyX25hbWU="), " Student");
TextView textView = this.f1550;
textView.setText(Base64Decode("SGkg") + string + Base64Decode("LCB5b3UgaGF2ZSBzdHVkaWVkOiA="));
this.f1553 = C0435.m1906((Context) this);
this.f170.setText(m1246(Integer.toBinaryString((int) Math.round((double) this.f1552.getInt(Base64Decode("dG9kYXl0aW1l"), 0)))));
double d = (double) this.f1554.getInt(Base64Decode("dG90YWx0aW1l"), 0);
```

Group 2

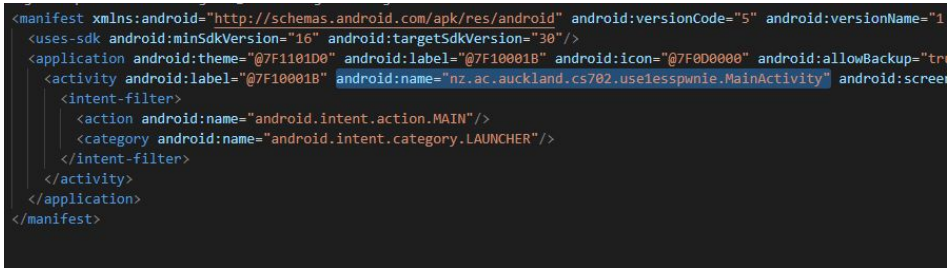
Once the source code was obtained, we were able to find the Main Activity Class file which lead us to the start of the Code flow. There was a lot of Junk data which made things hard to read. The techniques like **rename obfuscation** and **code flow redirection** along with the adding of **junk data** made the code hard to understand but not entirely intelligible. We assume this was done manually at first and then by using tools like minify and Proguard

```
package com.example.kakuropuzzler;

import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2131427359);
  }

  public void showCredits(View paramView) {
    startActivity(new Intent((Context)this, CreditsActivity.class));
  }

  public void showKakuro(View paramView) {
    startActivity(new Intent((Context)this, KakuroBoard.class));
  }

  public void showTutorial(View paramView) {
    startActivity(new Intent((Context)this, HowToPlayActivity.class));
  }
}
```



Group 5

- An application that decrypts and encrypts strings
- Initially challenging when trying to locate the MainActivity file, but with the right tools (fernflower, JADX and androguard) I was able to locate it
- Techniques used:
  - **Code obfuscation** most likely using built in minifyEnabled option in Android studio
  - **Control flow obfuscation** which made it very difficult to make sense of the logic of the code.



Group 6

Group 6's apk could only be assessed by using JADX for MainActivity, and FernFlower for ManagerActivity.

Group 6 utilised name overloading, such as:

```
class ManagerActivity->class a
class ManagerActivity->class a->void a(ArrayList<String> arrayList) {...}
class ManagerActivity->String[] a(ArrayList<String> arrayList) {...}
class ManagerActivity->void a(String str, ArrayList<String> arrayList) {...}
class ManagerActivity->String a(long[], int) {...}
```

Group 6 also globalised many variables within classes to hide their true use.

```
public class ManagerActivity extends c {
    public static String m = "Copied to Clipboard";
    static int n = 1;
    ArrayList<String> k;
    ArrayAdapter l;
    private Button o;
    private String p;
    private byte[] q;
    private Button r;
    private ListView s;
    private int t;
    private com.example.myapplication.a.a u;
    private String[] v = null;
    private String[] w = null;
    private int x;
```

Lastly, nested method calls made it difficult to find the true program flow

(Such as this example of creating password arrays from a SQLite query)

```
list2.add(0, com.example.myapplication.b.c.a(b3.a.a(b3.a.b(com.example.myapplication.a.a.b).intValue()), e));
while (iterator.hasNext()) {
    final String s2 = iterator.next();
    final String a8 = com.example.myapplication.b.c.a(b3.a.a(b3.a.b(s2).intValue()), e);
    list2.add(s2);
    list2.add(a8);
}
```

We did not notice any UI/application string encryption being used.