DEPRODEXCATOR: A TOOL FOR DECOMPILING AND DEOBFUSCATING PROGUARD AND DEXGUARD OBFUSCATED ANDROID APPLICATIONS

Anjum Ashraaf
QuantumRonics
Islamabad, Pakistan
anjumashraf@quantumronics.com

Hasham Sarwar
QuantumRonics
Islamabad, Pakistan
hashamsarwar@quantumronics.com

Ghulam Murtaza
Quantum Information and
Communication (QIC) Group
Quaid-e-Azam University Islamabad, Pakistan
azarmurtaza@hotmail.com

Kashif Raheem
National University of Sciences
and Technology (NUST)
Islamabad, Pakistan
krahim.dphd18seecs@seecs.edu.pk

ABSTRACT

Software code is a valuable asset for developers and organizations, but its protection from theft has become increasingly important in the digital age. Code obfuscation is somehow significant for protecting software against hacking, reverse engineering, and piracy, which can result in significant financial losses for software developers and businesses. Code obfuscation can help software owners safeguard their intellectual property, However, it can also enable malicious actors to conceal malware within the code. Code obfuscation does not prevent reverse engineering, rather, it complicates the process of understanding the code. This document presents a solution for deobfuscating the Proguard and DexGuard obfuscation techniques applied to Android Java Applications. A tool named DeProDexcator is developed which can decompile Android Java applications and deobfuscates applications obfuscated by ProGuard and DexGuard. DeProDexcator takes an APK as input, decompiles it, and offers various options to the user to analyze the decompiled code and deobfuscate the control flow and layout obfuscation techniques applied by ProGuard and DexGuard through static code analysis. Through identification of the correct program flow and analyzing the code through code comparison and efficient searching methods, DeProDexcator can effectively deobfuscate ProGuard and DexGuard-protected Android applications.

***K*eywords** Code Obfuscation, Reverse Engineering, Deobfuscation, Static Code Analysis

1 Introduction

Android is an open-source operating system based on top of the Linux kernel and is commercially sponsored by Google. Android was specifically developed for smartphones and tablets which now has become the largest adopted operating system for smartphones. Android uses APK (Application Package Kit) as software packages that are distributed by different platforms like Google Playstore and Samsung Galaxy Store [2]. APK contains application source code, resources, metadata, and instructions to install it on an android device. APKs are variants of the JAR (Java Archive) file format. Android installs this package into the device and source code and resources are then readable to the android device. Android applications are generally developed in IDE like Android Studio and Eclipse using Java or Kotlin. Once the application is developed the IDE puts everything into a container which is then named APK. [3] Since IDE compiles the code and put resources, metadata, and compiled code into APK. Android's easy-to-reverse engineer behavior makes applications vulnerable to reverse engineering. Whereas, code obfuscation methods are used to make it harder for reverse engineers to fully reverse engineer the applications. Code obfuscation is a method of mod-

ifying code by adding some roundabout expressions and redundant logic to make it more complex without affecting its execution behavior [15]. Android application developers use code obfuscation as a tool to make their applications secure against reverse engineering by protecting the main logic of the program. Whereas, it is also used by malicious application writers as a tool to hide malware inside applications. Obfuscation can not restrict someone from reverse engineering or decompiling the application rather it makes the code more complex so that it does not remain easy for them to understand. Different obfuscation methods are used to obfuscate android applications like Variable renaming, Layout obfuscation, Control flow obfuscation, Dummy code insertion, and String encryption are some commonly used obfuscation methods. Android Studio provides free access to its built-in obfuscator named ProGuard. ProGuard is an open-source java class shrinker, optimizer, and obfuscator [4]. ProGuard specifically reduces the size of the application by resource shrinking using the R8 compiler and uses variable renaming and layout obfuscation methods. DexGuard is another obfuscator that is commercially available. It applies ProGuard obfuscation along with control flow obfuscation and various other obfuscation methods. On the other hand, deobfuscation is used to make obfuscated patterns easy to understand. Deobfuscation refers to making obfuscated code more understandable by removing redundant logic, unused code removal, and removing control flow and layout obfuscation. This paper proposes a system named DeProDexcator that can reverse engineer android applications, analyze obfuscation results and deobfuscate ProGuard and DexGuard obfuscated applications. The proposed tool decompiles android applications and gets source code and resources out of them. It provides tools for analyzing the obfuscated code by effective searching, comparison, and static analysis of code. The proposed tool deobfuscates ProGuard and DexGuard obfuscated applications by removing control flow and layout obfuscation methods.

2 Preliminary Work

In [1], author Moses Yoni in their paper proposed the deobfuscation of android applications through static and dynamic code analysis. Dynamic analysis refers to running android applications in a controlled environment and monitoring the state of the application and operating system. During analysis, if the application is trying to access sensitive information or shows suspicious behavior it can be discovered. While static analysis is purely code analysis which analyzes the flow of the code and function calls inside the code. Static analysis can analyze code that can not be analyzed with dynamic analysis. For example, if the code is protected by some mechanism or some malicious code is hidden inside the application it can be discovered with static analysis. To refrain someone from static analysis most common technique used is code obfuscation which is mainly used to hide the functionalities inside the code and sometimes to hide the malicious code. The obfuscator targeted by the author is DashO by preEmptive, a widely used obfuscator for android applications. Variable renaming, string encryption, and reflection are the major obfuscation techniques used by the DashO obfuscator. The main focus of the author was on deobfuscating string encryption and reflection obfuscation by DashO. The deobfuscation pattern includes the combination of static and dynamic analysis of code and finding modules and functionalities inside the code.
Code obfuscation can be used to protect cyber-physical systems (CPS) from attacks by making it harder for attackers to understand the code and find vulnerabilities. This is particularly important for systems that control critical infrastructure, such as power grids or transportation systems, where a cyber attack could have serious consequences. However, code obfuscation is not a silver bullet solution for securing CPS. While it can make it harder for attackers to understand the code, it does not prevent all types of attacks. Attackers can still use other techniques, such as side-channel attacks or fuzzing, to find vulnerabilities in the system. In [8]. authors have discussed cyber-physical systems and modeling and simulation challenges for these systems from an operational security perspective.
In [5]. authors have proposed a method for deobfuscating string encryption obfuscated applications. The proposed method's focus is on the decryption of strings encrypted by any obfuscator so that they can be transformed into a readable form. Since strings inside the code are important and meaningful so string decryption makes the code more understandable. The proposed method by the author performs string decryption without an encryption key just by modifying Dalvik VM to dynamically extract the code when the application is running. The deobfuscation flow involves decompiling obfuscated Dex, obtaining the smali code, running on the Dalvik VM, and dynamically extracting the bytecode. String values returned from extracted code are then compared with the decompiled code with smali code parts application is repackaged with a returned string inserted into it. The proposed method is purely static analysis of code focusing on string encryption.
In [6]. authors proposed a new method for deobfuscating control-flow obfuscation by DexGuard. The proposed method detects the control flow of DexGuard-protected applications and restores the original code flows. The authors implemented a new deobfuscation tool specifically for DexGuard-protected applications. Three different levels of control flow obfuscation were discussed in this paper out of which two were easy to deobfuscate through ReDex [16]. Level 3 was deobfuscated by the tool developed by the authors. The focus of this paper is on control flow deobfuscation of DexGuard-obfuscated android applications and APKtool [17] is used for the decompilation.
In [9]. authors proposed a deobfuscation method for proguard which involves matching the obfuscated code with the

```
buildTypes {
    release {
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile(
        'proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

Figure 1: Enable ProGuard in Android Studio

non-obfuscated code already present in the database. Authors used software similarity algorithms like simHash and n-gram [18]. The source of non-obfuscated code includes open-source libraries and previously analyzed code. The proposed design primarily targets variable renaming code obfuscation. For reverse engineering of android applications, Apktool and smali [17] is used by the authors.
In [10]. authors proposed a method for the deobfuscation of ProGuard obfuscated android applications through probabilistic learning of big code and this code is used for training a model. The new unseen applications are deobfuscated by the model which is trained for deobfuscating proguard obfuscated code. The proposed idea by the author specifically targets the layout obfuscation technique by proguard and is implemented into an application named DeGuard.

3 Overview

This section describes a brief overview of reverse engineering and deobfuscation. The obfuscators and their obfuscation techniques targeted by DeProDexcator are also described in this section.
Software reverse engineering is defined as the process of analysis of system components and their inter-relationships and the creation of an end product in another form at a higher level of abstraction whereas it can be applied from any level of software development not necessarily the end product [13]. Software Reverse Engineering is an act of dismantling any software to see how it actually works generally for the purpose of analyzing and gaining knowledge [14].
Deobfuscation is removing the obfuscation techniques from decompiled code. It is referred to as analyzing the code and making it more formatted and understandable. It uncovers the original functionality of the code and results in a well-formatted code [19].

3.1 ProGuard

ProGuard is a generally used free Java obfuscator available to obfuscate android java applications. It generally applies the variables renaming obfuscation. It renames the original names with semantically obscure ones, making it hard for reverse engineers to understand. ProGuard also reduces the size of applications by resources shrinking [4]. ProGuard applies dummy code insertion obfuscation on some android control like SnackBar along with the layout obfuscation but generally, it only applies the layout obfuscation on all other android application controls. It can not obscure the code that is part of Android API or some external library as it would affect the execution behavior of the application. ProGuard is applied in android studio by just enabling its methods. Fig. 1 shows a code snippet of how to enable proguard in android studio java applications.

3.2 DexGuard

DexGuard is a commercially available java obfuscator and optimizer to obfuscate android applications with a higher level of obfuscation which makes it harder for reverse engineers to fully reverse engineer them. DexGuard provides all functionalities of ProGuard along with some other obfuscation techniques like control flow obfuscation, reflection, and arithmetic obfuscation making code more optimized and harder to understand. It segregates the code files into different folders and renames files to make it harder for deobfuscation. DexGuard optimizes code by removing redundant code, metadata, unused resources, and native libraries [7].

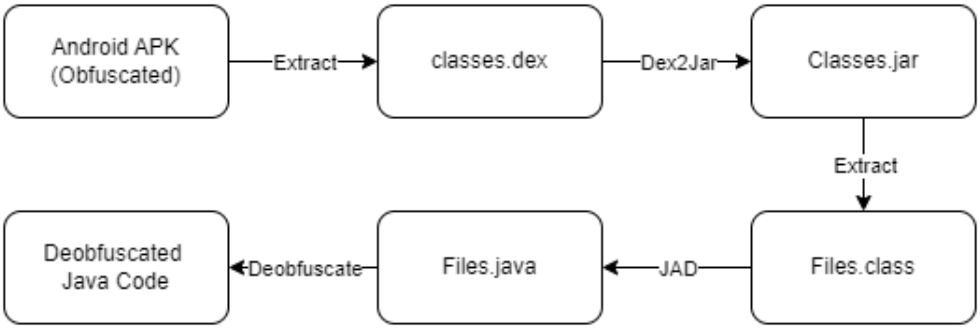

Figure 2: DeProDexcator Flow Diagram

4 DeProDexcator

This section describes an overview of DeProDexcator and its functionalities. Design and implementation are described in the next section of the paper.
DeProDexcator is a tool for the decompilation and deobfuscation of android java applications that take an APK as input, fetch classes.dex file from within the APK, and converts classes.dex file into a jar file using dex2jar [11] after the jar file is fetched it gets the classes files from the jar file and converts those classes files into Java files using java decompiler (JAD) [12]. JAD decompiles the java bytecode given as a class file into a readable form which is then structured using DeProDexcator. The tool makes the decompiled code intended according to coding conventions and colorizes the code parts into different colors and makes it more understandable. The code once decompiled is then stored in different folders and it sets everything for deobfuscation. DeProDexcator provides the features like searching for a specific keyword in the whole decompiled project, Keywords highlighting, comparison of different files, identifying classes, functions, and variables, and putting them into a tree form to make it more comprehensible. Fig. 2 shows how DeProDexcator works provided APK as input. DeProDexcator targets ProGuard and DexGuard java obfuscators and deobfuscates the control flow obfuscation applied by them.

Fig. 3 illustrates the flow design of deobfuscating java code given a java obfuscated file as input and outputs a deobfuscated java code.

5 Design and Implementation

This section describes the design and implementation of DeProDexcator. The tool involves reverse engineering of android java applications and deobfuscation of ProGuard and DexGuard android files. A specific Graphical User Interface(GUI) is developed for the decompilation and deobfuscation of android applications. The tool has two different components one is decompilation while the other one is deobfuscation. These components and their features are discussed in detail in this section. Python as a programming language is used for the implementation of the tool and GUI is developed with Tkinter.

5.1 Decompilation

Android uses an APK file to install it into the device. APK is an application package that includes the application's source code, resources, and metadata. Android applications can be reverse-engineered with the tools available like Apktool [17] that can decompile those APKs. DeProDexcator also decompiles the APKs along with the Deobfuscation. While decompiling obfuscated applications it makes sure that complete source code and resources are fetched from the application. During compilation Android java code is first compiled into class files which are java bytecode, java bytecode is then converted into Dalvik executable classes.dex file by Dexer which is part of Android SDK [21]. In decompilation, the APK file is first extracted and classes.dex [20] file is fetched from it. This classes.dex (Dalvik Executable) file contains all the source code from the application that is compiled into Dalvik bytecode. This Dalvik bytecode runs under Dalvik Virtual machine. Dalvik bytecode is decompiled into class files using dex2jar [11] which results in java bytecode. Java bytecode is not readable, it is then decompiled into java code by Java Decompiler (JAD) [12]. DeProDexcator decompiles all the Dalvik executable code into java code, making it ready for deobfuscation.

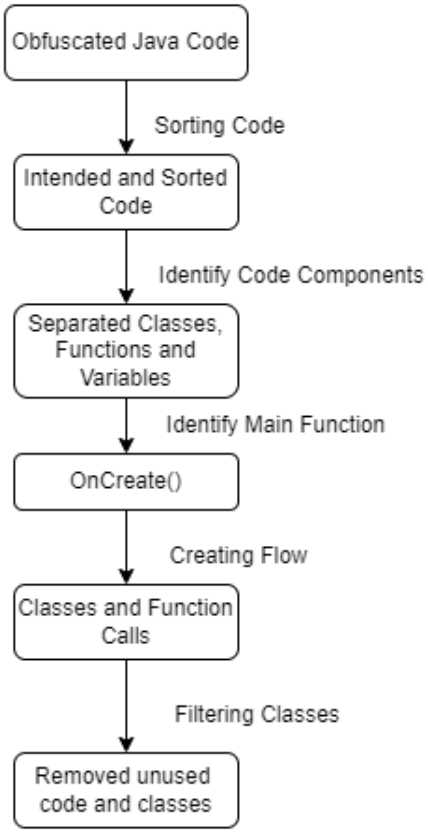

Figure 3: Deobfuscation Flow Diagram

5.2 Deobfuscation

Decompiled code is non-executable and thus can not be executed directly after decompilation. Obfuscation makes the code more complex by adding some arbitrary logic and renaming. DeProDexcator deobfuscates the code obfuscated by ProGuard and DexGuard. DeProDexcator deobfuscates code by sorting files, code structuring, searching and comparison, and identifying the code components from the decompiled code i.e packages, classes, functions, and variables. Separation of code into components makes it easy for static analysis. These features are discussed here in this section.

5.2.1 Files Sorting

When an android application is extracted it pulls out all the files and data from it. DeProDexcator sorts the files into respective folders to make it easy for understanding the whole project structure. Since DexGuard alters the original names of the files to special characters and puts the code into different folders. The tool sorts all the files according to the project structure and makes it more explicit.

5.2.2 Code Structuring

Since the decompiled code can not be executed directly and is more like a text file opened in any text editor, to help make the code more comprehensible DeProDexcator applies the code indentation and identifies the code components and colorizes them into different colors i.e different colors for variables, functions, classes, and packages. Code colorizing makes code more conceivable and helps build a better understanding of the code.

5.2.3 Searching and Comparison

DeProDexcator features searching and comparison of code. Searching within a single file and a whole project helps deobfuscating java files. DeProDexcator searches for specific input in the whole decompiled project and gets all the

```
protected void onCreate(Bundle bundle)
{
        super.onCreate(bundle);
        setContentView(0x7f0b001d);
        bundle = (Button)findViewById(0x7f08015c);
        tv1 = (TextView)findViewById(0x7f080084);
        Spinner spinner = (Spinner)findViewById(0x7f08018d);
        spinner.setOnItemSelectedListener(this);
        ArrayAdapter arrayadapter = new ArrayAdapter(this, 0x7f0b0069, tasbeehat);
        arrayadapter.setDropDownViewResource(0x1090009);
        spinner.setAdapter(arrayadapter);
        bundle.setOnClickListener(newObject()
        class _anm1
        {
        }
        );
}
```

Figure 4: Decompiled Non-Obfuscated Code

files containing the searched content and highlights the matched code parts. The tool also compares java code files, making obfuscation patterns very clear to understand. While comparing two different files DeProDexcator highlights the unmatched code parts which proves helpful in analyzing the obfuscated code.

5.2.4 Code Analysis

Code Analysis plays an important role in the deobfuscation of applications. DeProDexcator identifies all packages, classes, functions, and variables from within a java class and all these code components are then separated. The tool identifies the dummy code elements by calculating the number of times each element is called within or outside the class. For example, a variable is declared inside a class function but not called anywhere is a dummy variable. Programs flow is detected with the help of static analysis of decompiled code. DeProDexcator identifies the program flow given the main class as an input, it identifies classes, functions, and variables from the main class and within each class, it identifies the same code elements and creates a flow in the form of a tree which makes the code less complex and easy to understand. Fig. 9 below shows the flow detection of a DexGuard obfuscated decompiled application by DeProDexcator. ProGuard mainly applies the renaming obfuscation and thus makes static analysis easy while deobfuscation. Fig. 4 below shows the non-obfuscated decompiled java code which contains a function from the main class of java application and Fig. 5 below shows an obfuscated by ProGuard version of the same function. DexGuard applies control flow obfuscation, arithmetic obfuscation, reflection, and dummy code insertion along with the ProGuard obfuscation and code optimization. Fig. 6 below shows the obfuscated code version of the same function with DexGuard. DeProDexcator deobfuscates control flow obfuscation of DexGuard by identifying the right code flow of the program.
Fig. 7 shows a code snippet from the same application where two activities are called from an event inside a class. ProGuard shows the exact names of the classes and places them in the same directory where they belong, making it easy to understand and deobfuscate. Fig. 8 shows the same event from a DexGuard obfuscated application where two activities are called from that event and DexGuard applies the ProGuard obfuscation along with the renaming of classes and puts those classes in a separate folder and hides the actual names of those classes with some special characters. DeProDexcator deobfuscates the techniques applied by DexGuard and identifies the class names and links those classes with the main file. Fig. 9 shows the complete flow of the program starting from the main file in the application identifies the classes, functions, and variables inside all the classes, and links them with the main file which makes it deobfuscated.

6 Results

This section shows the results from DeProDexcator, including non-obfuscated and Obfuscated codes from ProGuard and DexGuard obfuscated android applications. The results show the difference between original,proguard, and dexguard obfuscated applications. Different obfuscation techniques applied by these obfuscators are discussed in this section.
Fig. 4 shows decompiled non-obfuscated code part from an android application which is a start function in any android activity. Decompiled non-obfuscated code is very similar to the original code with very few changes in it. Fig. 5 shows the decompiled ProGuard obfuscated version of the same android application's code. ProGuard applies the

```
public final void onCreate(Bundle bundle)
{
        super.onCreate(bundle);
        setContentView(0x7f0b001d);
        Button button = (Button)findViewById(0x7f080156);
        r = (TextView)findViewById(0x7f080087);
        Spinner spinner = (Spinner)findViewById(0x7f080187);
        spinner.setOnItemSelectedListener(this);
        bundle = new ArrayAdapter(this, 0x7f0b0067, p);
        bundle.setDropDownViewResource(0x1090009);
        spinner.setAdapter(bundle);
        class a
        {
        }
        button.setOnClickListener(new a());
}
```

Figure 5: Decompiled ProGuard-Obfuscated Code

```
protected void onCreate(Bundle bundle)
{
        super.onCreate(bundle);
        setContentView(0x7f09001c);
        bundle = (Button)findViewById(0x7f07005c);
        _fld02CF0971 = (TextView)findViewById(0x7f07002a);
        Spinner spinner = (Spinner)findViewById(0x7f070074);
        spinner.setOnItemSelectedListener(this);
        ArrayAdapter arrayadapter = new ArrayAdapter(this, 0x7f090035, _fld02CA0971);
        arrayadapter.setDropDownViewResource(0x1090009);
        spinner.setAdapter(arrayadapter);
        bundle.setOnClickListener(newObject()
        class _anm3
        {
        }
        );
}
```

Figure 6: Decompiled DexGuard-Obfuscated Code

rename obfuscation technique on code. Fig. 6 shows the decompiled DexGuard obfuscated version of the same android application's code. DexGuard applies the ProGuard obfuscation technique along with more code optimization. Fig. 7 shows a code snippet from the ProGuard obfuscated android application. Fig. 8 shows a code snippet from the DexGuard obfuscated same android application. In Fig. 9 a complete flow of a DexGuard obfuscated application is shown starting from the main activity it shows all the variables, functions, and class objects used inside that class, and from each class object, it recursively creates a flow by exploring every class inside the main class which identifies the dummy classes and provides the meaningful code.

```
public boolean onOptionsItemSelected(MenuItem menuitem)
{
        switch(menuitem.getItemId())
        {
                default: return super.onOptionsItemSelected(menuitem);
                case 2131230945: Toast.makeText(this, "Namaz Timings", 0).show();
                return true;
                case 2131230944: startActivity(new Intent(getApplicationContext(),
                com/example/tasbeeh/Duaa));
                Toast.makeText(this, "Duaa", 0).show();
                return true;
                case 2131230943: startActivity(new Intent(getApplicationContext(),
                com/example/tasbeeh/MainActivity2));
                return true;
        }
}
```

Figure 7: Classes Segregation by ProGuard

```
public boolean onOptionsItemSelected(MenuItem menuitem)
{
        switch(menuitem.getItemId())
        {
                case 2131165249:
                startActivity(new Intent(getApplicationContext(), o/FE9C));
                return true;
                case 2131165250:
                startActivity(new Intent(getApplicationContext(), o/01C0));
                Toast.makeText(this, "Duaa", 0).show();
                return true;
                case 2131165251:
                Toast.makeText(this, "Namaz Timings", 0).show();
                return true;
        }
        return super.onOptionsItemSelected(menuitem);
}
```

Figure 8: Classes Segregation by DexGuard

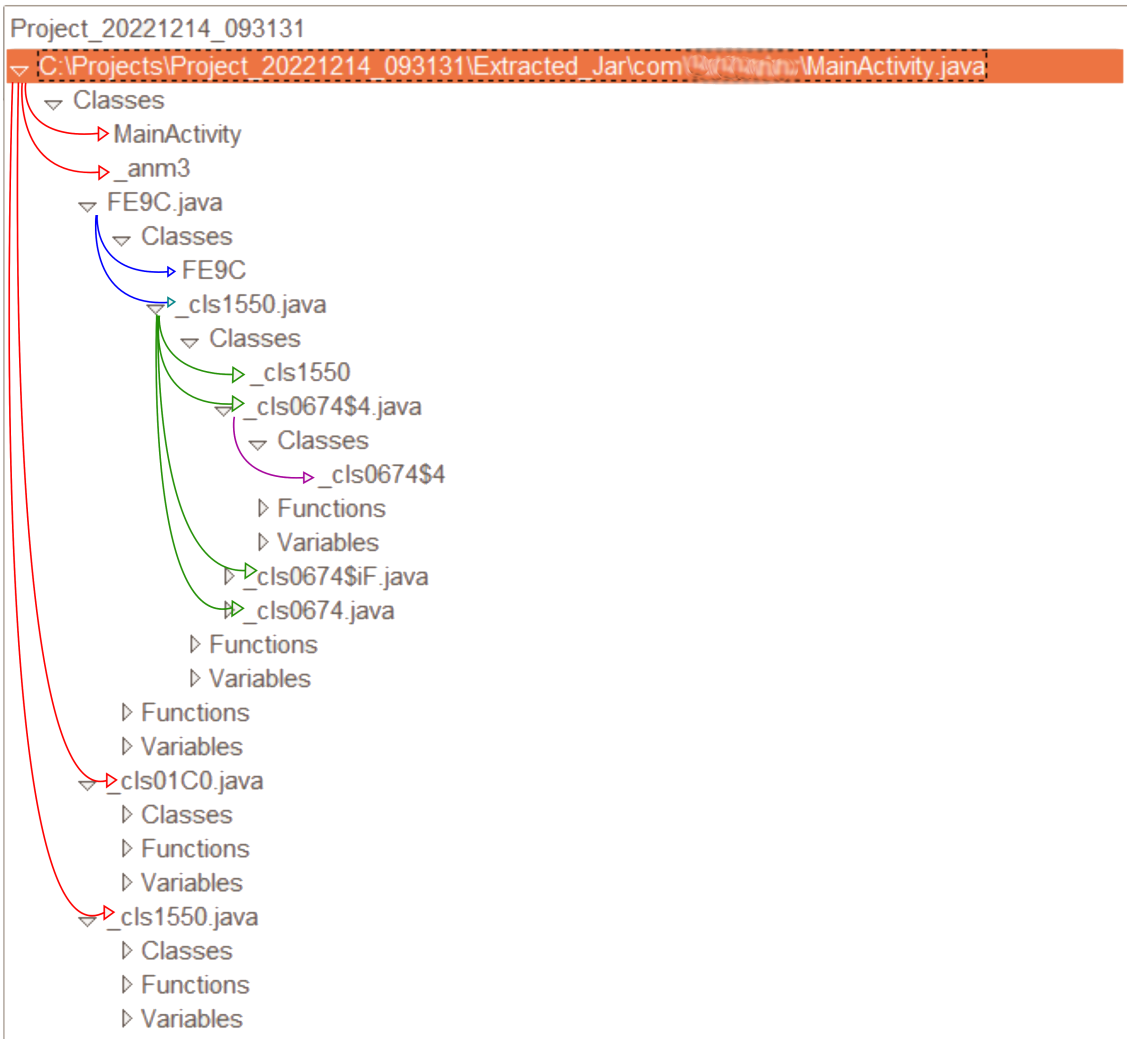

Figure 9: DexGuard DeObfuscation Flow

7 Future Work

Code obfuscation in android APKs is not a safe solution as it provides security through obscurity and hides the actual program logic but does not restrict anyone to reverse engineer the applications. However, code obfuscation can help in situations where other solutions like encryption and anti-viruses are not applicable. IoT devices are such examples where resources are so limited that encryption and malware detection mechanisms are not possible so, code obfuscation can play a role in somehow protecting these systems. DeProDexcator deobfuscates the control flow and layout obfuscation methods. We deobfuscated DexGuard obfuscated android applications by creating the right code flow and eliminating the unnecessary code from the application. Our tool can reverse engineer the APKs by decompilation and deobfuscation. We aim to deobfuscate DexGuard applications completely by removing reflection and arithmetic obfuscation in the future. A mechanism can be developed to recompile the applications after their complete deobfuscation and make them executable once again.

References

[1] Moses, Yoni. "ANDROID APP DEOBFUSCATION USING STATIC-DYNAMIC COOPERATION." (2018).

[2] "Android (Operating System)." Wikipedia, 30 Nov. 2022, en.wikipedia.org/wiki/Android (operating system).

[3] "What Is an APK File and What Does It Do." https://www.makeuseof.com/, www.makeuseof.com/tag/what-is-apk-file. Accessed 2 Dec. 2022.

[4] "ProGuard Manual: Home — Guardsquare." guardsquare.com, www.guardsquare.com/manual/home. Accessed 2 Dec. 2022.

[5] WooJong Yoo, Minkoo Kang, Myeongju Ji, and Jeong Hyun Yi. 2019. Automatic string deobfuscation scheme for mobile applications based on platform-level code extraction. Int. J. Ad Hoc Ubiquitous Comput. 31, 3 (2019), 143–154. https://doi.org/10.1504/ijahuc.2019.100730.

[6] G. You, G. Kim, S. Han, M. Park and S. -J. Cho, "Deoptfuscator: Defeating Advanced Control-Flow Obfuscation Using Android Runtime (ART)," in IEEE Access, vol. 10, pp. 61426-61440, 2022, doi: 10.1109/ACCESS.2022.3181373.

[7] "ProGuard Vs. DexGuard: An Overview — Guardsquare." GuardSquare, www.guardsquare.com/blog/dexguard-vs.-proguard. Accessed 7 Dec. 2022.

[8] K. Rahim and H. Khaliq, "Modeling and Simulation Challenges for Cyber Physical Systems from Operational Security Perspective," 2021 International Conference on Cyber Warfare and Security (ICCWS), Islamabad, Pakistan, 2021, pp. 63-69, doi: 10.1109/ICCWS53234.2021.9703029.

[9] Baumann, Richard & Protsenko, Mykolai & Müller, Tilo. (2017). Anti-ProGuard: Towards Automated Deobfuscation of Android Apps. 7-12. 10.1145/3099012.3099020.

[10] Bichsel, Benjamin, Veselin Raychev, Petar Tsankov and Martin T. Vechev. "Statistical Deobfuscation of Android Applications." Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016): n. pag.

[11] "GitHub - Pxb1988/Dex2jar: Tools to Work With Android .Dex and Java .Class Files." GitHub, github.com/pxb1988/dex2jar. Accessed 15 Dec. 2022.

[12] Wikipedia contributors. "JAD (Software)." Wikipedia, 28 Aug. 2022, en.wikipedia.org/wiki/JAD (software). Accessed 15 Dec. 2022.

[13] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: a taxonomy," in IEEE Software, vol. 7, no. 1, pp. 13-17, Jan. 1990, doi: 10.1109/52.43044.

[14] Lutkevich, Ben. "Reverse-engineering." Software Quality, 10 June 2021, www.techtarget.com/searchsoftwarequality /definition/reverse-engineering. Accessed 15 Dec. 2022.

[15] "Obfuscation (Software)." Wikipedia, 18 Nov. 2022, en.wikipedia.org/wiki/Obfuscatio (software). Accessed 15 Dec. 2022.

[16] Redex · an Android Bytecode Optimizer. fbredex.com. Accessed 15 Dec. 2022.

[17] Apktool - a Tool for Reverse Engineering 3rd Party, Closed, Binary Android Apps. ibotpeaches.github.io/Apktool. Accessed 15 Dec. 2022.

[18] Simhash. matpalm.com/resemblance/simhash. Accessed 15 Dec. 2022.

[19] Ndichu, S., Kim, S. and Ozawa, S. (2020), Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement. CAAI Trans. Intell. Technol., 5: 184-192. https://doi.org/10.1049/trit.2020.0026

[20] "Dalvik Executable Format" Android Open Source Project, source.android.com/docs/core/runtime/dex-format. Accessed 16 Dec. 2022.

[21] Patil, Sachin. "Creating and Reading DEX File - Sachin Patil." Medium, 19 June 2018, medium.com/@sachpa/creating-and-reading-dex-file-965ecd8b3206.