

REVERSE ENGINEERING AND STATIC ANALYSIS OF OBFUSCATION
TECHNIQUES IN ANDROID MOBILE APPLICATIONSMuhammad Nawaz Khan¹, Arooj safdar², Jalal Ud Din³, Muhammad Akram Khan^{*4},
Mehrullah Soomro⁵^{1,2*,4,5} Department of Computer Engineering, BUITEMS, Quetta,³ Department of Mathematics, COMSATS University Islamabad Campus, Pakistan¹muhammadnawazk0099@gmail.com, ²aroojsafdar5102004@gmail.com, ³jalalkhilji954@gmail.com.^{*4}mohammad.akram@buitms.edu.pk, ⁵mehrullahsoomro@gmail.comDOI: <https://doi.org/10.5281/zenodo.21867760>**Keywords**Android, APK, reverse
engineering, obfuscation, static
analysis, JADX, APKTool,
malware analysis**Article History**

Received: 25 January 2026

Accepted: 10 March 2026

Published: 24 March 2026

Copyright @Author

Corresponding Author: *
Muhammad Akram Khan**Abstract**

Android application developers are adept at obfuscating their applications to prevent the revelation of malicious activities or to safeguard intellectual property rights, particularly through the use of static analysis tools. This study focuses on obfuscation and structural complexity techniques in real world Android applications through reverse engineering. A set of Android application packages (APKs) which covers 10 functional categories and high and low sized applications were decompiled and statically analysed using JADX, APKTool, APK Easy Tool and APK Editor Studio and a Microsoft Excel spreadsheet was used for frequency analysis and visualisation. Sixteen obfuscation and structural techniques were identified and used to analyze the data by manually inspecting decompiled source code as well as manifest files and resource structures. The results indicate that the most common techniques are Third-Party SDK/Ad-SDK Integration (55) and Code Hiding through Third-Party Libraries (52), followed by Background Services (49) and Resource-Based Obfuscation (45), with the traditional Identifier Obfuscation technique being comparatively less common (28). Comparative analysis between reverse-engineering resistance, security risk and functionality suggests that Code Hiding using Third Party Libraries and SDKs provides the most effective resistance to static analysis because it removes the functionality from the analyzed artifact, as opposed to simply hiding the already present functionality. The results indicate that, in recent years, application protection in the Android ecosystem has evolved from code level protection using a single technique to multi-level protection strategies that integrate external dependencies, runtime indirection and architectural complexity. These findings are significant for malware analysts, static analysis tool developers, and application security reviewers.

1. Introduction

Modern Software Development Platforms generate intermediate or byte code forms of human-written code and these can, in many cases,

be re-compiled back into a human-readable source code that is very close to the original source code. The structural property for a very large percentage of deployed software is exploited in reverse

engineering, and it is the basis of the present study [1].

Reverse engineering (also known as back engineering) consists of the decomposition of an aircraft, software, machine, architectural structure, etc. to obtain information regarding its design [2]. In most cases this is the dismantling of the individual parts of a larger artifact, decomposing the logic, structure and behaviour without the original design documentation. Reverse engineering, as a concept, is the process of analyzing the structure, function and operation of a product or system to discover the technological principles that underlie it [3]. In software, and particularly in compiled mobile applications, reverse engineering can be used to retrieve a "readable approximation" of an application's logic from its distributed binary form and is the basis for legitimate security auditing activities, as well as for intellectual property theft and malware repackaging when used inappropriately.

1.1 Reverse Engineering of Software

Generally, reverse engineering follows a number of steps: acquisition of the compiled representation of software; unpacking or disassembly; reconstruction of the higher-level representation (such as pseudo-code, decompiled source or assembly-level output); analysis of the recovered higher-level representation to answer a particular question about the behaviour of the software [2]. This difference is complicated on the Android platform, where the pipeline goes through a two-stage compilation: source code (Java or Kotlin) is compiled to Java bytecode and later translated to Dalvik bytecode in a classes.dex file, which is then interpreted by the Android Runtime (ART) instead of a standard Java Virtual Machine (JVM) [4]. In general, Android applications are easier to decompile than applications compiled in native machine code, as a lot more structural and type information is available in the Dalvik bytecode than in native machine code for the underlying architecture, which developers who don't want their apps decompiled will have to fight against with obfuscation (see Section 1.5).

1.2 The Android Application Package (APK)

File Structure

An APK (Android Application Package) is a file that is compressed into a .zip file and contains everything necessary to install and execute an Android program. It contains, in general, four directories (res, assets, lib and META-INF), and three files (AndroidManifest.xml, classes.dex and resources.arsc) [4].

The res directory contains the files of the Android resources and is mapped to the generated R class and given numeric identifiers by the build system. The assets directory is very similar but unlike res, subdirectories of arbitrary depth can be created and static files can be stored in any structure, making it a common place for payloads or configuration information to be embedded. The lib directory contains code compiled for the specific hardware platforms, usually as shared-object (.so) native libraries, into processor-specific subdirectories (armeabi, armeabi-v7a, x86, x86_64 and MIPS). The signing and integrity information for a package is contained in META-INF and is used to ensure that an APK doesn't contain any changes that might have been made since it was signed.

AndroidManifest.xml is the main configuration file of all Android applications - each APK has only one such file, which declares the application package name, its version, required permissions, constituent parts (activities, services, broadcast receivers, and content providers); classes.dex contains a Dalvik bytecode representation of all classes in the application, organized in such a way that the Android Runtime can load and execute them directly; resources.arsc contains the mapping between resource files and their numeric resource identifiers, and can be used to programmatically find specific resources. All of these directories and files together constitute the artifacts that are targeted by all the static-analysis and reverse-engineering tools utilized in this study, namely JADX, APKTool, APK Easy Tool and APK Editor Studio.

1.3 Mobile Applications and Operating System Platforms

Mobile applications (or "apps") are software programs that are implemented specifically for use on handheld devices like a smartphone or tablet as opposed to software that is running on a desktop or web browser [6]. There are two mobile operating systems that are dominant all over the world: Android and iOS, and Android has a bigger installed base in the world and thus a larger reverse engineering and security research surface [7].

Android is an open-source Operating System originally developed by the Open Handset Alliance, based on a modified Linux kernel and other open-source components, and sponsored by Google since its early days and purchased the company that created it in 2005 [9]. In September 2008, the first Android powered device hit the market and today, the platform is the Largest (by installed base) mobile operating system base in the world [9] covering over 190 countries. This is mainly due to the fact that Android is an open platform, has a large number of manufacturers making devices based on it, has wide community support and has a high level of customizability [9, 10].

The iOS system on the other hand is Apple's mobile operating system, which was originally announced by the then-CEO of Apple, Steve Jobs on January 9, 2007 and released later that year [11]. Since then, Apple has also been releasing new models of the iPhone and new versions of the iOS operating system every year, and by mid-2025, over three billion iPhones had been sold worldwide, placing Apple as the world's top mobile handset vendor in terms of number of units sold since 2023 [11], [12]. The iPhone is a tightly coupled ecosystem of hardware and software that includes a high resolution touch display, sensors, camera, and several different wireless communication standards [12].

1.4 Platform Focus and Market Share

This study focuses on Android apps and not iOS apps for three reasons: There are three reasons that this study focuses on Android apps instead of iOS apps: Android has a much bigger surface area for reverse engineering and security research than

iOS, in that it currently dominates the global smartphone market with more than 70% of all smartphones installed, by comparison, as opposed to about 27% for iOS, according to recent industry estimates [13]. Android's share of the global smartphone market as of mid-2025 was about 72.46%, while there were over three billion Android devices currently in use worldwide, and iOS' share of global smartphones was about 27.13%, with a majority of 58.03% in the U.S. alone [13]. Samsung accounted for around 58 million Android smartphones shipped representing 19.7% of the global smartphone market, while Apple shipped around 46.4 million iOS devices representing 15.7% of the global smartphone market in the second quarter of 2025 [13].

Second, the open architecture of Android and the Dalvik/ART bytecode format makes it much easier to be analysed using static-analysis tools than the compiled and sandboxed application model of iOS. Thirdly, for Android there are mature and readily available static analysis tools like JADX and APKTool, whereas no equivalent open tools exist for iOS due to the compiled ARM machine code of iOS and the more limited distribution and code-signing model for the platform [7, 8]. Combined these factors render Android as the more relevant and more tractable platform to investigate obfuscation and reverse-engineering resistance on the empirical level, which is the scope of the present study.

1.5 Code and Resource Obfuscation

Obfuscation is technique which makes program more difficult to understand but keeps the same behavior with respect to its functionality. There are two main uses for it: benign obfuscation, which is used to defend against reverse-engineering of the source code of an application; and malign obfuscation, which is used to conceal malicious activity from antivirus software and static-analysis tools [14]. Obfuscation is used by developers to make static code analysis harder and obscure the implementation details from the competition, reverse engineers or automated scanners. Despite the sophistication of obfuscation tooling, it is still found that there are

still many common programs that lack obfuscation, especially in new and popular platforms like Android that are plagiarized, repackaged, and redistributed without change [14].

1.6 Overview of Obfuscation and Structural Techniques

“Obfuscation” turns a program, data, or system into code that's hard for humans and tools to understand but doesn't alter its functionality. The most common types reported in literature are: name and identifier obfuscation, in which the name of variables, methods and classes are replaced by arbitrary and meaningless names, control-flow obfuscation, which changes the structure of the flow of instructions in the program in order to make it difficult to detect its execution path, string and data encryption, which encrypts sensitive data that is explicitly stored in the code, code packing and compression, which hides the actual code until it is executed, call indirection and reflection, which replaces direct method calls with indirect ones in order to make it harder to reconstruct the call graph during static analysis, and instruction substitution, which makes simple instructions more complex while retaining the same functionality [1], [14]. In addition to these code-level techniques, Android applications can also use structural and architectural techniques, such as using third-party libraries and SDKs, background services, broadcast receivers, intent-based execution, and multi-process architectures, which are not particularly aimed at hiding source code, but can make behavioural analysis more difficult by spreading execution across multiple components, different communication mechanisms, and multiple processes. In Section 3.6, a sixteen-technique taxonomy was developed and applied based on this distinction between code-level obfuscation and architectural/structural complexity, and is studied in an empirical way in Section 5.

1.7 Significance of the Study

This study is important for a number of reasons. It improves the existing knowledge in the field of

security by giving frequency based empirical representation of the set of obfuscation and structural techniques observed in real world Android apps, and not just a taxonomic or theoretical description of these techniques. It provides a hands-on comparison of popular static analysis and reverse-engineering tools: the JADX, APKTool, APK Easy Tool, and APK Editor Studio, in terms of their capabilities and limitations for tools that use varying levels of obfuscation. It also helps in promoting better practices for malware detection by enumerating the techniques that are best suited against current static-analysis tools, and hence those the next generation of tools would have to overcome if they were to be effective. Last, the result provides developers and application-security reviewers with a better, fact-based understanding of actual obfuscation-architecture combinations found in deployed software, which can help them make informed choices regarding application-security strategies.

2. Literature Review

Android applications have grown in numbers at an unprecedented pace, and with them, the threats to their security have increased, mainly because of malware. The attackers are adopting advanced strategies for hiding malicious activities and evading detection, including code obfuscation. Reverse engineering and static analysis are now important techniques for app analysis and security vulnerabilities detecting on Android. This section summarizes the current literature related to Android Malware Analysis, reverse-engineering techniques, obfuscation methods, and recent advances with machine learning and large language models (LLMs).

2.1 Static Analysis Approaches

There are various types of static analysis tools available for analyzing Android applications without running them. In order to analyse the APK files, few studies are related to the extraction of permissions, manifest files and source code from the APK file. To analyze Android applications and detect malware, one reverse-engineering framework was proposed in the paper

with a total of six steps, which has an accuracy of 84.3% [15]. In the same way, other research projects examined the AndroidManifest.xml and the code to discover suspicious permissions like SMS, contacts and location rights that malware apps generally use [16]–[18]. These methods all exhibit the fact that static analysis can be effective in detecting permission misuse and possible data leakage.

2.2 Machine-Learning-Based Detection

There are many machine-learning methods that have been used to enhance the accuracy of malware detection. Among the models tested, Decision Tree found to be 83% accurate and Bagging as 88% accurate, a study [19] introduced a tool called AndRev which extracts permissions from APK files and applies machine-learning algorithms. A different one extracted some features from 1500 APK files and formed a feature matrix to train the machine-learning models [20]. While these approaches prove that feature-based learning is a very effective way to enhance the detection of malware, the quality of the extracted features is vital to these efforts and obfuscation can have a direct negative impact on the quality of the features.

2.3 Reverse-Engineering-Based Analysis

Reverse engineering is a crucial technique for comprehending Android applications. Several studies decompiled APK files and analyzed the structure of the files to discover vulnerabilities in the files like exposed components, data leakage, intent spoofing etc. [21]. There were tools that facilitated the automation of vulnerability scanning and penetration testing such as Drozer. Other work also assessed reverse engineering tools and concluded that none of the tools were able to completely reverse engineer Android applications, particularly with obfuscation techniques applied [22] [23]. In addition, research with Android binaries has revealed the presence of recognizable artifacts in the build process and obfuscation that can assist in forensic analysis [24]. The results demonstrate the current shortcomings of available tools, and the need for better analysis techniques [25].

2.4 Obfuscation Techniques in Android Applications

Many techniques are used to obscure information and to conceal malicious code. Often used obfuscation techniques are code obfuscation, string encryption, reflection, and integration with native code. Research has demonstrated that obfuscation makes traces traceable in Android binaries and they can be utilized for analysis and malware attribution [24]. Secondly, reverse engineering attacks have been explored, along with possible countermeasures like string encryption and runtime checks [26]. But these also make it much more difficult to detect malware.

2.5 LLM-Based Malware Analysis

There are new developments that have brought Large Language Model (LLM) to the forefront of malware analysis. In one study, the authors built an LLM-based system known as MalParse that extracts summaries of functions, classes, and packages from APKs, which can distinguish between malware and benign apps with a high accuracy of about 77% [27]. Other works employed LLMs to extract semantic information for creating structured representations of software behaviour [28]. Moreover, LLM-based methods have been used to summarise the APK functionality and extract the possible malicious behaviour in the APK [29]. These techniques speed up analysis, but can be problematic for heavily obfuscated code.

2.6 Tools and Frameworks for APK Analysis

Androguard, APKTool and Drozer are widely used reverse engineering and static analysis tools. Research works show that these tools have variations in performance, execution time and output quality [22, 23]. However, in many instances, several tools will be needed to analyse a single file, and most tools are limited by how they cope with the more sophisticated obfuscation methods [25]. To date, there have been marked developments in Android malware detection and analysis, such as static-analysis methods that can detect permission misuse [16]–[18] and machine-learning methods that can boost detection accuracy [19] and [20] and reverse engineering that

can provide greater insight into application behaviour [21]. LLM based approaches boost the efficiency of analysis but still rely on the visibility of the code, which is severely constrained in obfuscated applications [27, 29]. Collectively, these pieces of work motivate the empirical, frequency-based approach taken in the present study, in which a wide variety of real-world Android apps were analyzed using the relative prevalence of code-level and architectural obfuscation techniques.

3. Methodology

Methodology is the systematic procedures and techniques which are used for fulfilling the aim of the research. It offers a structured approach to the study, which involves data collection, analysis, implementation, testing and evaluation. The methodology is clearly followed, making the research results reliable, valid, and repeatable. The methodology of this study was systematic to examine the problem of the study, design the analytical process and test the effectiveness. The methodology involved the following stages: Tool selection, APK collection and validation, reverse engineering, taxonomy development, frequency analysis and results reporting.

3.1 Tools

Tools are the software applications, programming environments and technological resources used in the research and development process. Choosing the right tools is critical to the efficient, accurate and effective completion of the project. The following tools were used throughout the study to enable the implementation and evaluation of the proposed research framework: JADX is a forward/backward engineering tool available as command-line and GUI for Windows, Mac OS and Linux [32]. To obtain readable representations of decompiled classes and project structure, JADX was employed to translate Dalvik bytecode into source code that is more similar to Java, and is accessible via a GUI for interactive code browsing and a command-line interface for batch processing of multiple samples [32]. The tool was effective for quickly verifying logic and for identifying API calls, but it was found that the

output quality decreased considerably for some obfuscated applications or native libraries, so for these cases, smali code was cross checked with the output of JADX for consistency. The decompiling and recompiling of the APK files was made easier using apk easy tool [34]. Aside from support for signing and zip-alignment of APKs before installation, it offered a graphical interface to lessen the need for complicated command-line operations when dealing with Android packages, and was employed to extract resources, view manifest files, and rebuild APKs after modification [34]. The tool was mainly used to test small changes in structure and at the same time also to check the integrity of the code after decompilation. Android application packages (APK) were decompiled and recompiled using APKTool via command-line interface, which transformed the APK file to readable resource files and smali code [33]. APKTool decodes the resources.arsc file, XML resources and manifest file, giving understanding into the overall build structure of the application [33]. After changing, applications were re-compiled and re-tested with the tool. APKTool has been very well utilized to learn about obfuscation methods, and to verify application logic at a much more structural level, and is considered as one of the must have tools for Android reverse engineering and static analysis [33]. Android APK files have been edited using an open-source, cross-platform and free software known as APK Editor Studio [35]. With its graphical interface, application resources, icons and manifest files could be edited, signed and optimized without using complex command-line operations, it supported rebuilding and re-signing of applications directly from the GUI, and the editing of APKs, signing and optimizing of resources [35]. The primary purpose of using APK Editor Studio was to make fast visual changes to the APK and to configure the various files. The main data organisation and analysis tool used in this project was the Microsoft Excel. Data were gathered in rows and columns, each data point represented by a cell, which allowed data to be entered and formulas to be applied to automatically perform calculations, thus minimizing the risk of errors in the analysis

process. Totals, averages, and conditional outputs were calculated using built-in functions such as SUM, AVERAGE, and IF, respectively, and the frequency and percentage distribution of the obfuscation and structural techniques identified were visualized through the use of the charting features of Excel (bar charts and pie charts), as described in Section 5 [36].

3.2 Research Design

The Research Design is the structure that is followed while collecting, analysing and interpreting the data in order to answer the

research questions. It describes the method and approach used to make sure that the study was carried out in a systematic manner and yielded reliable and valid findings. The research design was sequential, consisting of four phases: preparation (tool selection and preparation); APK processing (collection and reverse engineering of mobile APKs); analysis (technique identification of obfuscation and statistical analysis of coding patterns); and reporting (results compilation). These phases are illustrated overall in Figure 1 and are dependent on each other in the following way:

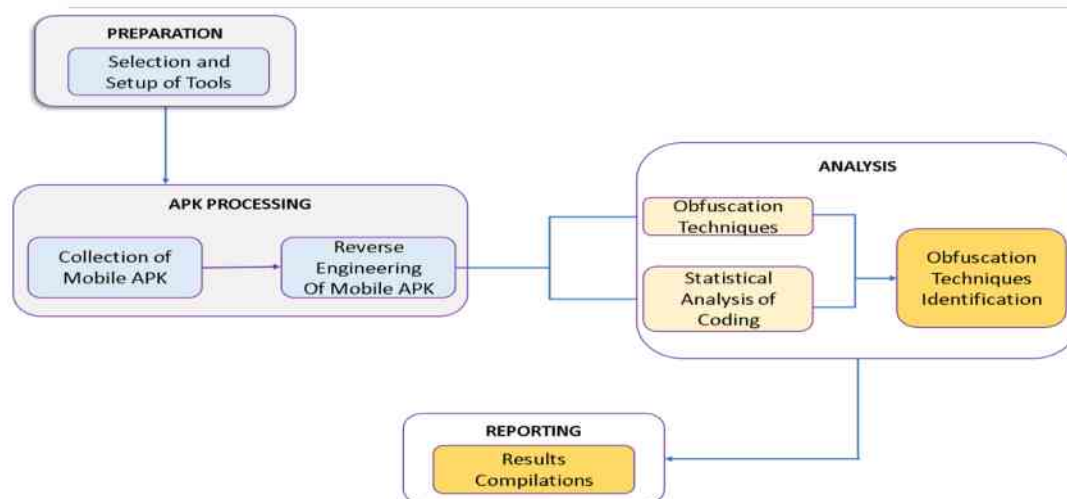


Figure 1: Research Methodology

The Preparation phase (tool selection and tool setup) leads to the APK Processing phase that involves the collection of the mobile APKs and their reverse engineering, as illustrated in Figure 1. The output of this stage is used in the Analysis stage where obfuscation techniques are detected concurrently with the statistical analysis of the decompiled coding patterns, both of which are consolidated in the Obfuscation Techniques Identification stage. After identification of the techniques, they are brought together in Reporting to create the final results compilation presented in Section 5.

3.3 Preparation and Tool Setup

Research objectives were defined at the beginning of the study, covering categories of obfuscation

found, as well as the measure of effectiveness of the analysis. The main tools that were used were APKTool, JADX, APK Easy Tool, APK Editor Studio and Microsoft Excel for organizing and visualizing the data, as mentioned in Section 3.1 [32]–[36]. A dedicated working environment was designed, consisting of the Android SDK and Android Debug Bridge (adb) [31] and a directory structure for raw APK files, decompiled output, results and logs. Documented versions and commands of the tools used to ensure reproducibility.

3.4 APK Collection and Validation

Before the data collection, the collection plan included the scope of the data such as number of applications and sources of applications. Only

legally obtained APK files used. Applications that were downloaded and relevant metadata such as package name, version, date and source information was captured for each of the samples. The files have been scanned for their integrity using checksums and digital signatures, and any file that is corrupted or tampered is not included in the dataset. Every sample was given a unique ID and a manifest record was created that included file paths and metadata. All raw APK files were kept in an archive in a read-only format to maintain the same samples for the entire duration of the study.

3.5 Reverse Engineering of APKs

All APK files were unpacked using APKTool [30] to get their resources, manifest and smali code, and the Dalvik bytecode was converted to readable source code with JADX [32] [33]. For those sections that required it, the smali code was directly checked to confirm and/or augment the decompiled output. For each sample, strings, certificates, native libraries, and assets were extracted. The declared permissions, exported components and manifest configuration information of each application were identified through static inspection based on AndroidManifest.xml schema specified in the official Android developer document [31]. Any manual observations not recorded in the various automated instruments were documented and added to the classification of techniques later.

3.6 Taxonomy of Obfuscation and Structural Techniques

Sixteen obfuscation and structural techniques were identified, categorized, and defined for this study, including identifier obfuscation, resource-based obfuscation, use of third party libraries, background services, excessive permissions, integration of third party SDKs, use of broadcast receivers, intent-based execution, component-based code separation, multi-activity structures, exported components, multi-process architecture, vendor-specific integration, custom application classes, push notifications and tracking, and usage of hardware features. The manually decompiled source code, manifest file and resource structure

were reviewed for each sample for the presence of evidence for each technique in the taxonomy, including non-descriptive or randomized identifiers, indirection resources, embedding of third-party libraries, declared background services, and requested permissions. The frequency data set (shown in Section 5) was created by recording each confirmed instance on the label of the technique that was observed.

3.7 Frequency Analysis of Identified Techniques

Microsoft Excel was used to count the occurrences of all sixteen techniques across the entire set of applications analyzed and create a frequency table with one row for each application and one column for each technique [36]. The total and relative frequency of each technique in the data set were computed using Excel's built in functions (SUM, AVERAGE, and conditional counting). With the frequencies of these techniques in mind, they were divided into the three categories outlined in Section 5.3: highly used techniques (more than 40 occurrences), moderately used techniques (20–40 occurrences), and less frequently used techniques (fewer than 20 occurrences). The distribution of absolute and percentage distribution of each technique were presented in the form of bar-chart and pie-chart respectively in excel. If the classification of a specific instance was unclear, the sample was re-inspected manually, then compared to the taxonomy outlined in Section 3.6, and a classification was determined. This analysis was descriptive and frequency based without any machine learning classifiers or clustering models were trained and no cross validated accuracy, precision or recall were calculated to be included in this study. Section 6.2 outlines one future research direction towards automating and learning classification of obfuscation techniques.

3.8 Identification, Labeling, and Reporting

The information collected in the reverse-engineering/inspection phase was integrated to identify techniques in each sample. Indicative criteria were created for each technique within the taxonomy (such as non-descriptive identifier names for identifier obfuscation, or the presence of declared background services in the manifest for

background-service usage), and applied uniformly throughout the dataset. Results in which the appropriate technique was not confidently determined were marked for manual review. For each application, a per-sample record was created that included a list of techniques detected and the evidence to support them (possible code snippets or manifest entries). In some cases, the name of a third-party library and/or SDK were known and identified, so as to aid in the classification of code-hiding and SDK-integration cases. Interrater reliability data (such as Cohen's kappa) was not computed because only one reviewer identified and labelled techniques. Interrater reliability data (e.g., Cohen's kappa) is not computed as this is acknowledged as a limitation of the classification process. Summary tables were created to show the number and proportion of each obfuscation and structural technique found throughout the set that correspond to the respective application categories discussed in Section 4.1. The frequency and percentage distribution of the techniques identified was plotted using the bar charts and pie charts and limitations, detection gaps, and possible sampling bias were noted, with some cases not being conclusively confirmed.

4. Experimental Setup and Dataset

The following section outlines the experiments used in the study. The main goal of these

experiments was to study Android applications in various categories and sizes, to remove the obfuscation from the applications using special tools and to evaluate the usefulness of static-analysis methods in understanding the structure and obfuscation mechanisms of applications. These experiments were intended to gather android applications from various categories, categorize them based on their characteristics, decompile the APK files, retrieve the important configuration files and analyse the structure of the applications. The results from these experiments give insight into application architecture, security mechanisms, permissions, and code-protecting techniques employed by Android application developers. A set of diverse applications was gathered and classified using functionality as the classification criteria to cover various application domains and development practices.

4.1 Dataset Categorization

A wide range of Android applications were gathered prior to performing reverse engineering and static analysis. Applications were grouped based on the applications according to functionality with the aim of covering various application domains and application development practices as shown in Tab. 1.

Table 1: Categories of Android Applications in the Experimental Dataset.

Category	No. of Applications	Category	No. of Applications
Games	15	Social	50
Photography	10	Education	12
Video Players	20	File Manager	10
Music	30	VPN	20
Health	15	Camera Apps	10
Fitness			8

In Table, you will find 11 categories of Android applications that are being presented, such as Games, Photography, Video Players, Music,

Health, Fitness, Social, Education, File Manager, VPN, and Camera applications. Of these, the most significant proportion of the dataset are Social

applications, then Music applications and then Video Players. The categorization process makes the research more reliable by adding applications with various functionalities, architectures, and security requirements. Because obfuscation methods are application domain specific, the study of multiple categories would give a broader understanding of the protection mechanisms used by Android applications: for instance, Social applications are more likely to access network resources, Camera applications require camera

permissions and multimedia processing capabilities, which differ from one another in the obfuscation and structural methods a developer would choose.

4.2 Selection of High-Sized APK Applications

The applications were categorized and then high-sized APK files were selected and analyzed. These applications have a lot of source code, various libraries, many resources and complex functions.

Name	Date modified	Type
(1)	10/26/2025 9:46 PM	File folder
(2)	10/26/2025 9:47 PM	File folder
(3)	10/26/2025 9:51 PM	File folder
(4)	10/26/2025 9:51 PM	File folder
(5)	10/26/2025 9:52 PM	File folder
(7)	10/26/2025 9:53 PM	File folder
(8)	10/26/2025 9:54 PM	File folder
(9)	10/26/2025 9:55 PM	File folder
(10)	10/26/2025 9:56 PM	File folder
(11)	10/26/2025 9:57 PM	File folder
(12)	10/26/2025 9:58 PM	File folder
(13)	10/26/2025 9:59 PM	File folder
(14)	10/26/2025 9:59 PM	File folder
(15)	10/26/2025 10:00 PM	File folder
(16)	10/26/2025 10:01 PM	File folder
(17)	10/26/2025 10:02 PM	File folder
(18)	10/26/2025 10:03 PM	File folder
(19)	10/26/2025 10:04 PM	File folder
(20)	10/26/2025 10:06 PM	File folder
(21)	10/26/2025 10:07 PM	File folder

Figure 2: High-Sized APKs Repository (Screenshot).

The repository of large Android apps selected for experiments is shown in figure 2. The architectures and code bases of these applications tend to be more complex than lightweight applications. APKs can be quite large and include several DEX files, third-party libraries, multimedia content, and advanced features. Therefore, they create a realistic scenario to study a complex application and the way reverse-engineering tools work on it. Furthermore, large applications often adopt more sophisticated obfuscation to ensure

sufficient protection of the IP and to discourage the analysis of unlicensed applications, and thus research of such applications can facilitate a consideration of the effect of code obfuscation on the reverse engineering process.

4.3 Selection of APK Applications

Besides large applications, small sized APK's were also chosen to provide a baseline comparison to the experiments.

Name	Date modified	Type
(4)	10/25/2025 11:24 PM	File folder
(5)	10/25/2025 11:25 PM	File folder
(6)	10/25/2025 11:26 PM	File folder
(7)	10/25/2025 11:27 PM	File folder
(8)	10/25/2025 11:27 PM	File folder
(9)	10/25/2025 11:28 PM	File folder
(10)	10/25/2025 11:29 PM	File folder
(11)	10/25/2025 11:29 PM	File folder
(12)	10/25/2025 11:31 PM	File folder
(13)	11/24/2025 11:28 PM	File folder
(14)	10/25/2025 11:32 PM	File folder
(15)	10/25/2025 11:33 PM	File folder
(16)	10/25/2025 11:33 PM	File folder
(17)	10/25/2025 11:35 PM	File folder
(19)	10/25/2025 11:36 PM	File folder
(20)	10/25/2025 11:37 PM	File folder
(21)	10/25/2025 11:38 PM	File folder
(22)	10/25/2025 11:39 PM	File folder
(24)	10/25/2025 11:41 PM	File folder
(25)	10/25/2025 11:42 PM	File folder

Figure 3: Low-Sized APKs Repository (Screenshot).

In Figure 3, the lightweight Android application are displayed as part of the data set. These apps usually have fewer number of classes, resources and libraries, and so the architecture is simpler and the package size is smaller. Low sized APKs are included to enable comparison of APK reverse-engineering results from varying complexity of applications. Smaller applications tend to have simpler structures, and can therefore be decompiled and analysed more efficiently, making

it easier to compare high sized applications with low sized ones to check if application size affects the implementation and effectiveness of obfuscation techniques.

4.4 Collection of Original APK Files

The next step was the collection of the original APK files, then followed reverse engineering activities.

Name	Date modified	Type	Size
(1)	10/25/2025 11:49 am	APK File	2,148 KB
(2)	10/25/2025 11:49 am	APK File	2,312 KB
(3)	10/25/2025 11:49 am	APK File	802 KB
(4)	10/25/2025 11:49 am	APK File	10,091 KB
(5)	10/25/2025 11:49 am	APK File	8,875 KB
(6)	10/25/2025 11:49 am	APK File	12,562 KB
(7)	10/25/2025 11:49 am	APK File	5,293 KB
(8)	10/25/2025 11:49 am	APK File	15,442 KB
(9)	10/25/2025 11:49 am	APK File	4,301 KB
(10)	10/25/2025 11:49 am	APK File	7,817 KB
(11)	10/25/2025 11:49 am	APK File	5,451 KB
(12)	10/25/2025 11:49 am	APK File	1,943 KB
(13)	10/25/2025 11:49 am	APK File	40,993 KB
(14)	10/25/2025 11:49 am	APK File	5,191 KB
(15)	10/25/2025 11:49 am	APK File	9,869 KB
(16)	10/25/2025 11:49 am	APK File	10,931 KB
(17)	10/25/2025 11:49 am	APK File	4,321 KB
(18)	10/25/2025 11:49 am	APK File	30,254 KB
(19)	10/25/2025 11:49 am	APK File	10,375 KB
(20)	10/25/2025 11:49 am	APK File	4,330 KB
(21)	10/25/2025 11:49 am	APK File	2,454 KB
(22)	10/25/2025 11:49 am	APK File	4,275 KB
(23)	10/25/2025 11:49 am	APK File	24,230 KB
(24)	10/25/2025 11:49 am	APK File	2,497 KB
(25)	10/25/2025 11:49 am	APK File	3,338 KB
(26)	10/25/2025 11:49 am	APK File	2,563 KB
(27)	10/25/2025 11:49 am	APK File	15,100 KB
(28)	10/25/2025 11:49 am	APK File	811 KB
(29)	10/25/2025 11:49 am	APK File	589 KB

Figure 4: Uncompiled (Original) APK Files.'

The figure 4 shows the APKs compiled as they are. The applications at this stage are Dalvik bytecode and resources that are packaged and cannot be understood by humans. The original APK files are important because they are the same applications that are distributed to end users; changes made prior to analysis may impact the validity of the experimental results. The original APKs give

insightful data on package structures, embedded resources, permissions, and security configurations.

4.5 APK Decompile Process

All the collected APK files have been decompiled to translate machine-readable bytecode into human-readable code and resources.

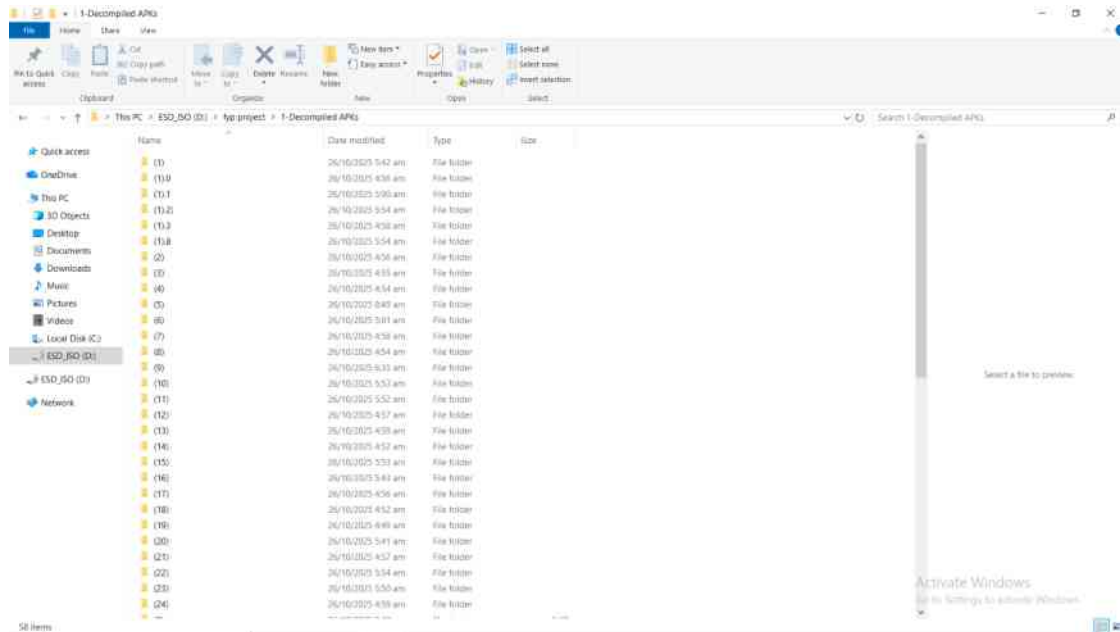


Figure 5: Decompiled APKs Interface.

The result of the decompilation process is depicted in figure 5. The APKs were turned into folders with the source code, XML resources, assets, and configuration files. One of the most crucial phases of Android reverse engineering is decompilation, which offers insights into the logic and functionality of the apps. The quality of the resulting decompiled output can also be used to

assess the obfuscation tools used: Strong obfuscation will yield less readable output, and analysis will be much more difficult.

4.6 Android Manifest File Analysis

AndroidManifest.xml files were extracted and analysed after the decompilation.

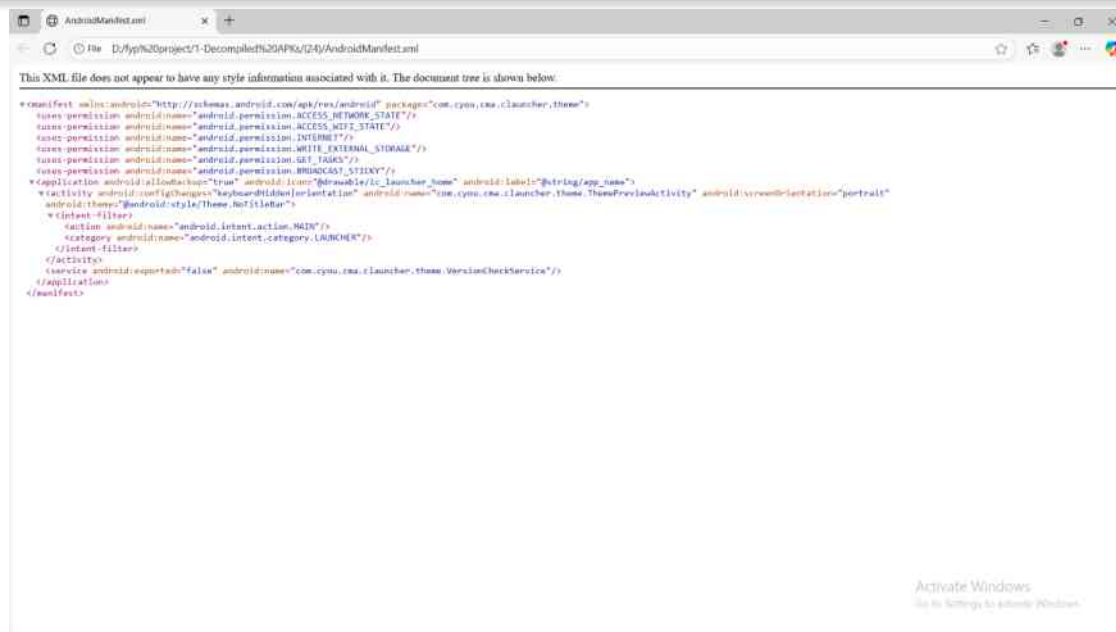


Figure 6: Extracted AndroidManifest.xml File.

The Android Manifest File of a sample app is displayed in figure 6. Android Manifest File is the main configuration file for all Android applications, which includes all of the package names, application components, permissions, services, receivers, activities, and requirement for the SDK [31]. Using manifest analysis,

permissions that a given application is asking for can be determined, and the functions of the app can be inferred. One of the key reasons that manifest analysis is significant in the field of security research is that it exposes an application's attack surface and security settings.

4.7 Analysis Using the JADX Tool

For deeper static analysis, the JADX reverse-engineering framework was used [32].

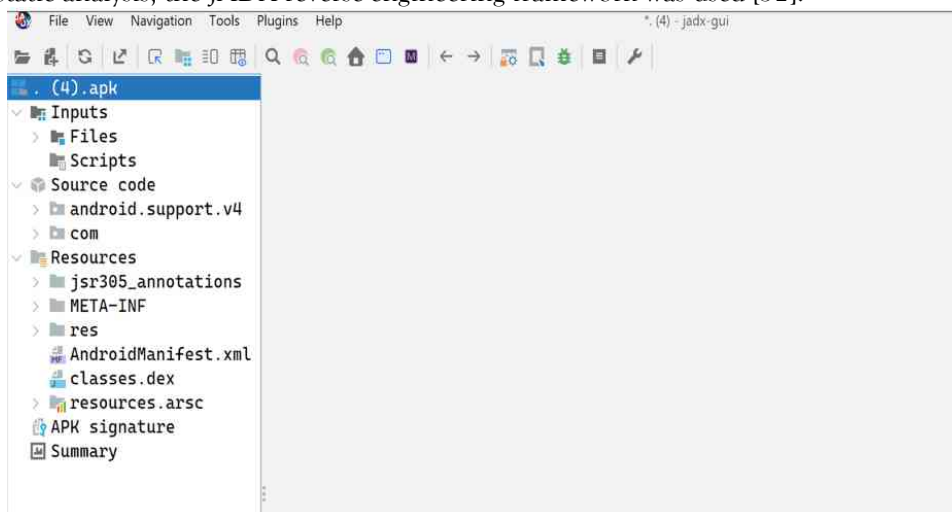


Figure 7: JADX-GUI Interface Showing APK Internal Structure.

The graphic interface of the JADX tool is shown in figure 7. The left side navigation displays the internal organization of the APK, such as source-code folders, AndroidManifest.xml, resources, and DEX files. JADX translates Dalvik bytecode into Java source code and thus makes Android applications easier to understand and analyze [32]. This tool is an essential component of this

research work since it offers an organized setting to analyze source code and to evaluate the efficiency of the reverse-engineering processes.

4.8 Detailed Manifest Inspection through JADX

This general analysis was then followed with a detailed analysis of the Android Manifest File with the help of the JADX interface.

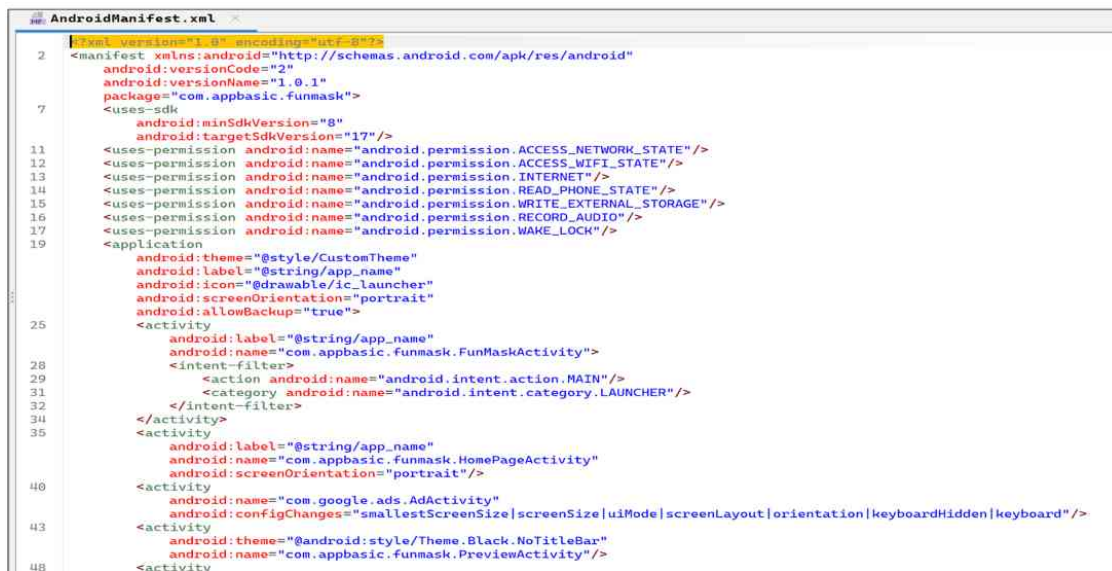


Figure 8: Detailed AndroidManifest.xml Inspection Within JADX.

As shown in figure 9, the Android Manifest File is opened in JADX. The structure of XML that is displayed is broken down into details of app permissions, activities, themes, SDK versions, and configuration settings. A number of permissions can be seen such as, INTERNET, ACCESS_NETWORK_STATE, WRITE_EXTERNAL_STORAGE, RECORD_AUDIO, and WAKE_LOCK. Detailed manifest inspection of this type aids in the correlation of the application's behaviour to the behaviour of its source code and also supports recognizing security-sensitive operations.

5. Results and Discussion

The results of the reverse engineering and static analysis of Android Applications are presented in this section. A total of 16 different obfuscation and structure techniques were found in the analyzed applications and all of them were

employed for enhancing the application logic, making the architecture more complex, making communication between the components easier and improving functionality.

5.1 Overview of Techniques

The results are presented in qualitative and quantitative analysis. First, the obfuscation and structural techniques identified are displayed in tabular format. The groups of techniques are then talked about in terms of occurrence and effect on reverse engineering. Finally, comparative analyses are provided by separating reverse engineering resistance, security risk and function.

5.2 Selected Technique Descriptions

We noticed 28 cases of identifier obfuscation in the examined applications. This technique maps useful class names, package names, methods, and variables into unreadable names, mainly because it

makes the code less readable, but can also make it more difficult for the original authors to maintain. This technique is very effective in protecting intellectual property, but can also cause problems for the original authors of the code. Resource-based obfuscation was detected 45 times, using indirect references to sensitive resources, encrypted resources, or strings, to make these values more difficult to detect in a static analysis, but less easy to read. The presence of code hiding via third party libraries and SDKs was observed 52 times and is one of the most common styles used: developers often bundle substantial functionality into third party libraries and SDKs, and important

application logic is not immediately apparent when analyzing the applications. Forty-nine background services were found, enabling applications to perform operations like synchronization, process notifications, track location, and gather data without involving the user, creating a covert execution path for the operations that makes behavioural analysis difficult. 41 instances of "excessive permission requests" were found in the permission analysis, where it can be that applications get access to sensitive device resources that they are not necessarily supposed to use for their actual purpose, leading to privacy and security risks.

5.3 Frequency of Identified Techniques

Technique	Frequency	Category
Third-Party SDK / Ad SDK Integration	55	Highly Used
Code Hiding via Third-Party Libraries	52	Highly Used
Background Services	49	Highly Used
Resource-Based Obfuscation	45	Highly Used
Excessive Permissions	41	Highly Used
Component-Based Code Separation	40	Moderately Used
Multi-Activity Structure	38	Moderately Used
Intent-Based Execution	36	Moderately Used
Broadcast Receivers	32	Moderately Used
Identifier Obfuscation	28	Moderately Used
Exported Components	15	Less Frequent
Hardware Feature Usage	14	Less Frequent
Multi-Process Architecture	10	Less Frequent
Push Notifications / Tracking	9	Less Frequent
Vendor-Specific Integration	8	Less Frequent
Custom Application Class Usage	6	Less Frequent

Third-Party SDK Integration (55 occurrences) was the single most common technique, closely followed and then by Code Hiding through Third-

Party Libraries (52) and Background Services (49), accounting for 107 out of 478 observations. In contrast, Custom Application Class Usage,

Vendor-Specific Integration and Multi-Process Architecture were the least common, indicating less use of specialized architectural techniques in the applications surveyed.

5.4 Obfuscation-Oriented versus Structural Complexity Techniques

Technologies are then distinguished based on whether they work to cover up the application logic (identifier obfuscation, resource-based obfuscation, code hiding through third party libraries) or simply add to the architecture complexity without directly hiding the source code (background services, broadcast receivers, intent-based execution, component-based code

separation, multi-activity structures, multi-process architecture). In terms of type of technique used for obfuscation, code hiding using third party libraries (52 occurrences) was much more prevalent than renaming identifiers (28). This aligns with previous research that has found that Android developers are more likely to hide functionality in third party libraries than in identifiers alone [30]. Structural techniques, on the other hand, hide the application behaviour by executing the same code in multiple components, a technique that we believe is more challenging to dynamically and behaviourally analyze than direct code obfuscation.

5.5 Reverse-Engineering Resistance and Security Risk

Technique	RE Resistance	Frequency	Basis for Resistance
Code Hiding via Third-Party Libraries/SDKs	Very High	52	Removes functionality from the analyzable artifact, creating a genuine black-box effect.
Multi-Process Architecture	Very High	10	Distributes execution across isolated process/memory boundaries.
Resource-Based Obfuscation / Indirection	High	45	Delays disclosure of sensitive values until runtime, defeating static inspection.
Background Services	High	49	Conceals execution paths that run without direct user interaction.
Intent-Based Execution	High	36	Replaces direct calls with indirect, event-driven communication.
Identifier Obfuscation	Medium	28	Renames identifiers but leaves logic and control flow fully traceable.

The most security critical techniques were Excessive Permissions (41 occurrences) and Exported Components (15 occurrences) – both sensitive resources can be exposed by excessive permissions and improperly configured exported components can enable external applications to call internal functions and/or access protected resources, a pattern that has been extensively reported as a frequent source of Android vulnerabilities.

5.6 Determination of the Most Effective Technique

When analyzed in terms of reverse engineering resistance, host application adoption, implementation costs, and risks added to the host application, Code Hiding through Third-Party Libraries and SDKs comes out on top as the most successful method for resisting reverse engineering in this data set. The difference from identifier obfuscation is that code hiding means that the

code is not part of the analyzable APK, not hidden by the names but by the complete absence of the code, unless an external library is available or can be decompiled. This is consistent with external literature; a significant percentage of obfuscation identified in large-scale studies of the Google Play Store is found in third-party libraries which developers do not explicitly use to obfuscate their applications [30]. In one survey, 43% of Play Store applications and 73% of third-party applications were reported as having been renamed [2] while another large scale investigation found up to 64% of third-party applications were renamed [4]. These numbers are similar to what has been found in the current study and indicate that dependence on external SDKs and libraries, and not intentional efforts to obfuscate by the developer, is the primary source of resistance to reverse engineering in today's Android applications. One striking absence is classical control-flow obfuscation, which, according to previous research, is more difficult to create on Dalvik than on traditional desktop platforms, because the Dalvik runtime places register-type restrictions on its control-flow obfuscations [5]. This implies that the dominance of structural and SDK-based techniques that emerged here may not be a reflection of a lack of the developer's will to obscure control flow but could be due to limitations of the tools that can be used on different platforms, which is a limitation of the taxonomy used here.

5.7 Practical Implications

In this study, code hiding via third party libraries and SDKs was the most resistant method observed, but is seldom used alone in the real world software. Most commercial code hardening solutions use a mix of resource-based obfuscation to expose embedded secrets, identifier renaming as a low-cost baseline transformation, and use of external SDKs, which provides additional protection when not used for code hardening. The multi-layered design is in line with the overall trend revealed in this study that Android application protection is now evolving from a single protection technique from a code obfuscation approach to a combination of

technique-based, architectural, resource-based and ecosystem-based techniques operating together.

6. Conclusion

In this study, an empirical, reverse-engineering based analysis of the 16 obfuscation and structural complexity techniques was conducted on a wide range of real-world Android apps. The outcomes indicate that the protection of Android applications has significantly moved beyond the mere renaming of identifiers and the hiding of basic resources to strategies increasingly reliant on third party SDK integration, indirection at runtime, and modulatory architectural designs. Overall, the most prevalent method was Third-Party SDK/Ad-SDK Integration (55 occurrences), while the strongest resistance to static analysis, which is to remove functionality from the analyzable artifact, was Code Hiding through Third-Party Libraries and SDKs (52 occurrences). Background Services and Resource Based Obfuscation were also commonly used, and included hidden execution paths and delayed disclosure of sensitive runtime values respectively. Structural techniques such as intent-based execution, broadcast receivers, component-based code separation, and multi-activity architecture did not make it impossible to extract source code directly, but made the process of control-flow reconstruction and behavioural analysis significantly more difficult. The analysis of the data identified excessive permissions and exported components as the major risk factors from a security perspective. Overall, the results suggest that reliance on third-party ecosystems, runtime and background execution mechanisms, modular architecture, and indirect data/control flow, are more influential on the complexity of Android applications and their resistance to reverse engineering than the intentional use of code obfuscation methods.

6.1 Future Work

There are several directions to extend this work: (i) the development of automated, machine-learning based obfuscation and structural complexity detection and classification engines at scale; (ii) the integration of dynamic analysis (runtime

monitoring, sandbox execution and system-call tracing) to account for behaviours that are not possible to see with static analysis, such as background services and SDK controlled execution; (iii) comparative analysis for identifying the obfuscation and complexity patterns in benign and malicious applications to isolate patterns that can only be used by malicious intent; (iv) the use of graph-based deep-learning models (Call Graphs, Control-Flow Graphs, Data-Flow Graphs combined with Graph Neural Networks or transformer architectures) for automated detection of patterns; (v) a study on the performance, energy and resource consumption trade-offs introduced by background services, multi-process architectures and SDK integrations; (vi) a study on the evolution of obfuscation and structural complexity across different Android platform versions and Google Play policy changes; and (vii) the development of quantitative, weighted security risk-scoring models, incorporating the patterns of permission usage, SDK integration, execution complexity and architectural design, for security analysts and application marketplace reviewers.

6.2 Limitations

There are a number of caveats to take into account when considering the findings of this study. No inter-reliability statistic computed because technique identification and labeling was carried out by one reviewer. There was no predictive analysis or machine-learning classifiers were not trained, and there was no cross-validated accuracy, precision, or recall metrics generated. The taxonomy was based on the existing literature, but classical control-flow obfuscation was not found as a prevalent pattern in the dataset analysed, which could be due to the practical difficulties of implementing complex control-flow obfuscation on Dalvik bytecode or simply to the fact that this technique is not used heavily in the overall Android ecosystem. Lastly, although the dataset covers 10 functional categories and high and low-sized applications, it can not be considered to be fully representative of the global Android application ecosystem, either with regard to

regional app stores or distributed applications that are not published through the Google Play Store.

REFERENCES

- M. Popa, "Techniques of program code obfuscation for secure software," *Journal of Mobile, Embedded and Distributed Systems*, vol. 3, no. 4, pp. 205–214, 2011.
- A. A. Aremo, *Reverse Engineering Research*. Campbellsville University, Dec. 2021. [Online]. Available: <https://doi.org/10.13140/RG.2.2.28030.51520>. Accessed: 2026.
- Wikipedia contributors, "Reverse engineering," https://en.wikipedia.org/wiki/Reverse_engineering, 2025. Accessed: Oct. 28, 2025.
- X. Zhang, F. Breiteringer, E. L  chinger, and S. O'Shaughnessy, "Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations," *Forensic Science International: Digital Investigation*, vol. 39-C, 2021.
- S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," *Lecture Notes in Computer Science*, 2018.
- Wikipedia contributors, "Smartphone," <https://en.wikipedia.org/wiki/Smartphone>, 2025. Accessed: Oct. 28, 2025.
- Ahmad, B., Jillani, S. A., Rafique, M., & Soomro, A. A. (2026, January). Design and Monitoring of a Tree-Inspired Vertical Axis Wind Turbine for Efficient Urban Wind Utilization. In *2026 1st International Conference on Innovations in Information and Communication Technologies (IICT)* (pp. 1-6). IEEE.
- C. G. Thomas and A. Jayanthila Devi, "A study and overview of the mobile app development industry," *International Journal of Applied Engineering and Management Letters*, vol. 5, pp. 115–130, 2021.

- GeeksforGeeks, "Introduction to Android development," <https://www.geeksforgeeks.org/android/introduction-to-android-development/>, Jan. 2025.
- M. R. Islam, M. R. Islam, and T. A. Mazumder, "Mobile application and its global impact," *International Journal of Engineering & Technology*, vol. 10, pp. 72–78, 2010.
- Wikipedia contributors, "iPhone – History," <https://en.wikipedia.org/wiki/IPhone#History>, 2025. Accessed: Oct. 28, 2025.
- J. Gonzalez-Sanchez and M. E. Chavez-Echeagaray, "iPhone application development," in *Companion to the 25th Annual ACM SIGPLAN Conference (SPLASH/OOPSLA '10)*, Reno/Tahoe, NV, USA, pp. 321–322, 2010.
- Backlinko, "iPhone vs Android user & revenue statistics (2025)," <https://backlinko.com/iphone-vs-android-statistics>, 2025. Accessed: Oct. 28, 2025.
- M. Conti, V. P., and A. Vitella, "Obfuscation detection in Android applications using deep learning," *Journal of Information Security and Applications*, vol. 70, p. 103311, 2022.
- Ahmad, B., Ali, S., Muneer, M., Fatima, A., & Abbas, N. (2025). Wind energy integration into the SAARC region: A comprehensive review of optimal wind sites for a sustainable super smart grid. *Wind Energy*, 3(2).
- I. G. A. Adnyana, P. G. S. C. Nugraha, and B. R. A. Nugroho, "Reverse Engineering for Static Analysis of Android Malware in Instant Messaging Apps," *Journal of Computer Networks, Architecture and High Performance Computing*, vol. 6, no. 3, pp. 1460–1469, Jul. 2024,
- S. M. Myat and M. T. Kyaw, "Analysis of Android Applications by Using Reverse Engineering Techniques," *International Journal of Innovative Science and Research Technology (IJISRT)*, vol. 4, no. 3, pp. 551–556, Mar. 2019.
- M. T. Kyaw, Y. N. Soe, and N. S. M. Kham, "Security analysis of Android application by using reverse engineering," in *Proc. Int. Conf., Yangon, Myanmar*, pp. 171–177, 2019.
- M. Patil and D. Pramod, "Predicting Unexpected Permission Authorization of Android Application Using Machine Learning," *Webology*, vol. 17, no. 3, pp. 238–261 2020.
- B. A. Gyunka, A. T. Oladele, and O. Adegoke, "Adaptive Android APKs reverse engineering for features processing in machine learning malware detection," *International Journal of Data Science*, vol. 4, pp. 10–25, June 2023.
- C. Happer, "Reverse engineering Android apps and testing with Drozer," *ResearchGate Preprint*, July 2025.
- T. Blazytko and M. Schloegel, "The future of reverse engineering with large language models," <https://synthesis.to/>, 2024. Accessed 2025-10-30.
- Ahmad, B., Majeed, M. K., Soomro, A. A., & Jillani, S. A. (2026, January). Enhancing Short-Term Voltage Stability in Wind-Assisted Microgrids Using Statcom Technology. In *2026 1st International Conference on Innovations in Information and Communication Technologies (IICT)* (pp. 1-6). IEEE.
- A. Jain, H. Gonzalez, and N. Stakhanova, "Enriching reverse engineering through visual exploration of Android binaries," in *Proc. Conf. on Android Security and Reverse Engineering*, Varanasi, India / Fredericton, Canada, 2025.
- A. Albakri, A. Ali, H. Fatima, A. Ali, M. Mohammed, N. M. Elzein, and A. Ahmed, "Survey on reverse-engineering tools for Android mobile devices," *Mathematical Problems in Engineering*, vol. 2022, pp. 1–7, Jan. 2022.

- S. Humbatov, M. Najafli, D. Guliyeva, and D. Amrahova, "Reverse engineering attacks on Android applications: Techniques, case studies, and defence strategies," *UNEC Journal of Computer Science and Digital Technologies*, vol. 1, no. 1, pp. 60-68, 2025.
- B. J. Walton, M. E. Khatun, J. M. Ghawaly, and A. Ali-Gombe, "Exploring large language models for semantic analysis and categorization of Android malware," in *Proc. Conf. on Cybersecurity and Artificial H. Siala and K. Lano, "Towards using LLMs in the reverse engineering of software systems to Object Constraint Language," in Proc. 32nd IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER 2025), 2025.*
- D. Sahoo, "Reverse engineering of APKs using LLMs," M.S. thesis, New York University, Tandon School of Engineering, May 2024.
- D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in Google Play," in *Proc. 34th Annual Computer Security Applications Conference (ACSAC 2018)*, San Juan, PR, USA, pp. 222-235, 2018.
- Google, "App manifest overview," Android Developers, <https://developer.android.com/guide/topics/manifest/manifest-intro>. Accessed: Aug. 7, 2026.
- Skylot, "jadx: Dex to Java decompiler," GitHub repository, <https://github.com/skylot/jadx>. Accessed: Aug. 7, 2026.
- C. Tumbleson and R. Wiśniewski, "Apktool: A tool for reverse engineering Android apk files," <https://ibotpeaches.github.io/Apktool/>. Accessed: Aug. 7, 2026.
- "APK Easy Tool," <https://www.apkeasytool.com/>. Accessed: Aug. 7, 2026.
- A. Gorishnyak, "APK Editor Studio," GitHub repository, <https://github.com/kefir500/apk-editor-studio>. Accessed: Aug. 7, 2026.
- Microsoft, "Microsoft Excel," Microsoft 365, <https://www.microsoft.com/en-us/microsoft-365/excel>. Accessed: Aug. 7, 2026.