UNEC Journal of Computer Science and Digital Technologies **Section:**
Volume 1, № 1, 2025, pp. 60-68 **Cybersecurity & Privacy**
https://doi.org/10.30546/UNECCSDT.2025.01.060

Reverse Engineering Attacks on Android Applications: Techniques, Case Studies, and Defence Strategies

Shamil Humbatov[1*], Murad Najafli[2], Dilbar Guliyeva[3], Dilbar Amrahova[4]

[1*] *Department of Digital Technologies and Applied Informatics, UNEC, Baku, Azerbaijan*
0000-0001-6487-6034, shamil.humbatov@unec.edu.az
[2] *0009-0007-3665-9623, najafli.murad.rovshan.2022@unec.edu.az*
[3] *0009-0009-4437-1499, guliyeva.dilbar.faiq.2022@unec.edu.az*
[4] *0009-0001-1210-1437, amrahova.dilbar.jeyhun.2022@unec.edu.az*

Abstract

Reverse engineering remains a critical threat to the security of Android mobile applications due to the platform's open-source nature and the accessibility of its application packages. This paper investigates the technical vulnerabilities that expose Android applications to reverse engineering, including the ease of decompiling APK files and extracting sensitive logic and data. It explores both static and dynamic analysis techniques, runtime manipulation, and code modification, which are commonly used by attackers to bypass security mechanisms or alter application behavior. Drawing on real-world case studies, the paper illustrates how these techniques have been exploited in practice, compromising application integrity and user privacy. In response, a range of defense strategies is evaluated, such as code obfuscation, string encryption, native code protection, root detection, and runtime integrity checks. The study also considers the legal and ethical implications of reverse engineering, emphasizing the importance of intellectual property protection and compliance with international regulations. The findings highlight the necessity of a multi-layered defense approach that integrates technical safeguards with legal awareness to effectively mitigate risks and enhance the resilience of Android applications.

***Keywords*:** android security, reverse engineering, mobile application protection, static and dynamic analysis, code obfuscation, runtime manipulation

Received:	*Revised:*	*Accepted:*	*Published:*
29/05/2025	*03/06/2025*	*06/06/2025*	*14/06/2025*

1. Introduction

Reverse engineering is the process of analyzing an existing product, software, or system to understand its structure, functionality, and operational principles [1]. While traditionally used in industrial and technological domains, this approach has become increasingly prevalent in the field of mobile applications. In the context of mobile apps, reverse engineering involves decompiling compiled code and internal resources to convert them into a human-readable format [2, 3]. This technique can be employed by both security professionals and malicious actors for different purposes.

For security analysts and researchers conducting penetration testing on mobile applications, reverse engineering techniques are valuable tools for identifying vulnerabilities, assessing application architecture, and evaluating security levels [4]. However, these same techniques are often misused. Attackers can exploit reverse engineering to uncover server endpoints, APIs, encryption methods, cryptographic keys, and other sensitive data embedded within the application [5]. In some cases, the entire logic of the application is exposed, enabling cloning or malicious modifications [6-7].

The open-source nature of the Android platform and its extensive tool support make it attractive to both developers and attackers. While openness fosters innovation, it also introduces additional security risks [8]. Therefore, ensuring resilience against reverse engineering is a critical aspect of secure mobile application development.

This study aims to identify the vulnerabilities that make Android applications susceptible to reverse engineering attacks, examine the techniques used to carry out such attacks, and evaluate the effectiveness of various defense strategies. The findings are intended to help mobile app developers and security professionals better understand real-world threats and implement effective countermeasures.

2. Vulnerabilities of Android Applications to Reverse Engineering

One of the primary reasons Android applications are vulnerable to reverse engineering is the transparent and easily analyzable structure of their application packages (APK). Android apps are typically developed using high-level programming languages such as Kotlin or Java and are compiled into Dalvik Executable (.dex) files within the APK. These .dex files can be decompiled back into readable Java code using publicly available tools, such as JADX and APKTool, enabling attackers to examine the application's internal logic, data structures, and security mechanisms with relative ease [3], [8].

An APK file is essentially a ZIP archive that contains various components, including AndroidManifest.xml, classes.dex, and directories such as res/, lib/, and assets/. During the reverse engineering process, these files can be extracted and analyzed to reveal the application's structure and behavior [3]. The classes.dex file, in particular, holds the core logic and functionality of the application, and its decompilation can provide detailed insights into how the application operates.

In many cases, developers do not apply code obfuscation techniques, leaving variable and method names in their original, human-readable form. This significantly simplifies the reverse engineering process. Moreover, storing sensitive information such as passwords or API keys in plaintext within the code introduces critical security vulnerabilities, potentially leading to unauthorized access and data leakage [7].

Although the decompiled output often includes Smali code—a lower-level representation compared to Java—it still provides sufficient detail for understanding application behavior, execution flow, and data handling. This makes reverse engineering not only feasible but also highly effective for attackers with moderate technical expertise [9].

Reverse engineering techniques are generally categorized into two main types:

1. **Static Analysis**: This involves examining the application's code without executing it. Analysts inspect the source or binary code to identify vulnerabilities, understand control structures, and extract functional logic. Static analysis is useful for uncovering hardcoded secrets and insecure coding practices [4].
2. **Dynamic Analysis**: In this approach, the application is executed in a controlled environment, and its behavior is monitored in real time. Techniques such as runtime monitoring, debugging, tracing, and profiling are employed to observe how the application interacts with system resources and responds to user inputs. Dynamic analysis is particularly effective in identifying runtime vulnerabilities and behavioral anomalies that static methods may overlook [5].

Together, these vulnerabilities and analysis techniques highlight the importance of implementing robust security measures during the development of Android applications to mitigate the risks posed by reverse engineering.

3. Reverse Engineering Attack Techniques and Tools

APK Decompilation: Android applications are distributed in the form of APK (Android Package Kit) files, which encapsulate essential components such as classes.dex, AndroidManifest.xml, resources, and metadata. These files can be extracted and analyzed using reverse engineering tools like APKTool and JADX, which allow attackers to decompile the application and access its internal code structure [3], [8].

For example, a simple Kotlin class responsible for checking user login status might appear as follows after decompilation:

```
class LoginCheck {
    fun isLoggedIn(): Boolean {
        return false
    }
}
```

An attacker can easily modify this logic by changing return false to return true, thereby bypassing authentication checks and gaining unauthorised access to the application. This type of manipulation demonstrates how reverse engineering can compromise application logic and security with minimal effort.

Such vulnerabilities highlight the importance of implementing code obfuscation and runtime integrity checks to prevent unauthorised code inspection and modification [9].

Code Modification and Application Modding: Once an application has been decompiled, attackers can manipulate its source code to alter core functionalities. This technique, commonly referred to as *modding*, is frequently used in mobile games and premium applications to bypass restrictions or unlock paid features without authorisation. By modifying specific logic within the code, attackers can gain unfair advantages such as unlimited in-game currency, access to locked content, or removal of advertisements [2], [7].

For instance, consider the following Kotlin class that manages a user's virtual currency:

```
class GameManager {
    var coinCount = 0

fun addCoins(amount: Int) {
    coinCount += amount
}

fun getCoins(): Int {
    return coinCount
}
}
```

An attacker could modify the `getCoins()` method to always return a large value, such as:

```
fun getCoins(): Int {
    return 999999  // permanently inflated coin count
}
```

This alteration disrupts the in-app economy and undermines the monetization model of the application. Such unauthorized modifications not only violate intellectual property rights but also pose significant security and ethical concerns. The widespread distribution of modified APKs can lead to revenue loss for developers and expose users to additional risks, including malware embedded in unofficial app versions [7].

Runtime Attacks (Frida, Xposed): In contrast to static code modification, runtime attacks allow adversaries to alter an application's behaviour during execution without modifying the underlying codebase. These attacks are typically carried out using dynamic instrumentation frameworks such as **Frida** and **Xposed**, which enable real-time hooking and manipulation of application logic [9].

For example, consider a Kotlin function that checks whether a user has premium access:

```
fun isPremiumUser(): Boolean {
    return false
}
```

Using Frida, an attacker can override this method at runtime without altering the APK file. The following script demonstrates how the method can be intercepted and forced to return true, thereby bypassing access controls:

```
Java.perform(function () {
    var PremiumChecker =
Java.use("com.example.app.PremiumChecker");
    PremiumChecker.isPremiumUser.implementation = function ()
{
        console.log("Premium status bypassed using Frida");
        return true;
    };
});
```

This technique enables attackers to manipulate application logic dynamically, granting unauthorized access to premium features or restricted content. Because the original code remains unchanged, such attacks are more difficult to detect through static analysis alone. Runtime manipulation poses a significant threat to application integrity and highlights the need for robust runtime protection mechanisms, such as integrity verification, anti-hooking techniques, and behavior monitoring [8].

Exploitation via Rooted Devices: Rooting an Android device grants the user, or any application running on the device, elevated privileges that bypass the standard security model enforced by the operating system. This process enables access to system-level files and configurations that are otherwise protected. While rooting can be useful for advanced users and developers, it significantly weakens the device's security posture and opens the door to various forms of exploitation [9].

Attackers can leverage rooted environments to bypass application sandboxing, extract sensitive data, and perform advanced dynamic analysis. For instance, if an application stores authentication tokens using Android's Shared Preferences mechanism as shown below:

```
val prefs = context.getSharedPreferences("userdata",
Context.MODE_PRIVATE)
val token = prefs.getString("auth_token", null)
```

On a rooted device, these files can be accessed using tools like **ADB (Android Debug Bridge)** or file explorers with root access. As a result, sensitive information such as auth_token can be retrieved and misused without triggering any security alerts [8].

The implications of such attacks extend beyond unauthorized access. They can compromise user privacy, damage the application's reputation, and result in significant financial losses for developers. Rooted device exploitation is particularly dangerous because it allows attackers to operate beneath the application layer, often evading traditional security mechanisms.

Effective countermeasures—such as root detection, encrypted storage, and runtime integrity checks—are essential to mitigate these risks. These defense strategies are discussed in detail in Section 5 of this paper.

4. Real-World Examples and Attack Cases

Reverse engineering techniques are not merely theoretical—they have been actively used in real-world scenarios to compromise mobile applications. The following case studies illustrate how attackers exploit reverse engineering to bypass security mechanisms, access premium features, and compromise user privacy.

Clash of Clans: Deceptive Mod APKs: In popular mobile games such as *Clash of Clans*, unauthorized modified versions of the application—commonly known as **Mod APKs**—are widely distributed. These versions often grant users unlimited in-game resources, such as gems and gold, or unlock premium features without payment. Typically, these modified apps operate either offline or on private servers disconnected from the official Supercell infrastructure. While they may offer enhanced capabilities, they do not reflect the authentic gameplay experience and are often unstable or misleading [2].

Moreover, many of these Mod APKs are embedded with malicious code, posing significant risks to user data and device security. In response, Supercell has implemented strict policies prohibiting the use of modified clients and may permanently ban accounts attempting to connect to official servers using such unauthorized versions.

Medium: Circumventing Paywalls via Reverse Engineering: The publishing platform *Medium* employs a subscription model that restricts access to certain articles. In 2018, security researcher Yuval Shprinz demonstrated how reverse engineering could be used to bypass these restrictions. By decompiling the Medium Android application, he analyzed how the app verified subscription status [10].

Shprinz intercepted and modified HTTPS requests by disabling certificate pinning, allowing him to manipulate API calls and falsely present himself as a subscriber. This enabled him to access paywalled content without authorization. The incident revealed a critical overreliance on client-side validation and prompted Medium to strengthen its backend authentication mechanisms [10].

This case underscores the risks of relying solely on client-side logic for enforcing access controls and highlights the importance of secure communication practices and robust server-side validation in mobile application design.

WhatsApp: Privacy and Security Risks in Modified Versions: WhatsApp, one of the most widely used messaging platforms globally, has also been targeted by reverse engineering. Several unofficial variants—such as **YoWhatsApp**, **GBWhatsApp**, and **WhatsApp Plus**—have emerged, offering features not available in the official app. These include hiding read receipts, viewing deleted messages, spoofing last seen status, and managing multiple accounts on a single device [7].

These modified versions are typically created by reverse engineering the original application and injecting new functionalities. However, they often compromise end-to-end encryption, collect user data without consent, and may contain malware. Meta, the parent company of WhatsApp, explicitly prohibits the use of such apps and may suspend accounts

found using them. These cases underscore the dual-edged nature of reverse engineering—while it can enable feature expansion, it also introduces serious security and privacy vulnerabilities [6], [7].

5. Defence Mechanisms Against Reverse Engineering

To ensure the security of mobile applications, it is essential to implement robust defense mechanisms against reverse engineering attacks. These attacks allow adversaries to analyze application code, exploit vulnerabilities, and gain unauthorized access to premium features or sensitive data. Several advanced techniques have been developed to mitigate these risks [8], [9]:

Code Obfuscation (ProGuard, R8): Code obfuscation is a widely used technique to make source code difficult to understand. In the Android ecosystem, tools such as **ProGuard** and its more advanced successor **R8** are employed to rename classes, methods, and variables into meaningless symbols, thereby concealing the application's logic. For example, functions like loginUser() or makePayment() may be transformed into a1() or b2(), making reverse engineering significantly more challenging for attackers [9].

String Encryption: Sensitive information such as API keys, URLs, and credentials stored in plaintext can be easily extracted during decompilation. **String encryption** addresses this issue by storing such data in an encrypted format. At runtime, the application decrypts the strings as needed. For instance, an encrypted URL might appear as a base64-encoded string like "`U29tZUVuY3J5cHRlZFVST...`", which is unintelligible without the decryption logic. This approach enhances data confidentiality and complicates static analysis [8].

Native Code Protection (NDK): Since Java and Kotlin code can be easily decompiled, critical application logic can be implemented using the **Native Development Kit (NDK)** in C or C++. Native code is more resistant to reverse engineering due to its binary format and the complexity of analyzing compiled native libraries. This technique is particularly effective for protecting cryptographic operations and licensing checks [8].

Root Detection: Applications running on rooted devices are more vulnerable to tampering and data extraction. Implementing **root detection** mechanisms allows applications to identify such environments and respond accordingly—either by restricting functionality or terminating execution. For example, some banking apps refuse to run on rooted devices to prevent unauthorized access to sensitive financial data [8].

Anti-Debugging Techniques: To prevent attackers from analyzing application behavior during runtime, developers can implement **anti-debugging** techniques. These include checking for active debuggers using methods like android.os.Debug.isDebuggerConnected() or introducing time-based checks and silent crashes to disrupt debugging attempts. Such measures increase the difficulty of dynamic analysis and delay reverse engineering efforts [8].

Runtime Integrity Checks (Checksum, Signature Validation: Applications can verify their own integrity during execution by checking the **checksum** of their APK file or validating their **digital signature**. If any unauthorized modifications are detected—such as tampering with the code or replacing resources—the application can refuse to run. This ensures that only the original, unaltered version of the app is executed, thereby protecting against code injection and repackaging attacks [8].

6. Legal and Ethical Perspectives

Developing a mobile application is a complex process that requires not only technical expertise but also significant intellectual effort and creativity. Developers invest substantial time and resources into building secure, functional, and user-friendly software. However, the growing prevalence of **modified APKs (Mod APKs)**—unauthorized versions of applications altered through reverse engineering—poses a serious threat to this effort [7], [10].

These modified applications often appeal to users by offering premium features for free, removing advertisements, or enabling hidden functionalities. Despite their popularity, such practices are neither legally permissible nor ethically justifiable. They undermine the intellectual property rights of developers and violate the terms of service of the original applications [1].

Technical and Legal Implications

The creation of Mod APKs typically involves decompiling the original APK, modifying its code, and redistributing the altered version. This process constitutes a direct violation of software integrity and is legally recognized as an infringement of intellectual property rights. In many jurisdictions, such actions are prohibited by law and may result in civil or even criminal penalties [10].

International Legal Frameworks

- **United States**: The **Digital Millennium Copyright Act (DMCA)** prohibits the circumvention of digital rights management (DRM) systems and considers the creation and distribution of Mod APKs a violation of copyright law [11].
- **European Union**: Under the **InfoSoc Directive**, software is granted strong legal protection. Modifying and redistributing applications without consent is treated as a breach of licensing agreements and may lead to fines or imprisonment in some member states [4].
- **Germany**: The **Urheberrechtsgesetz (UrhG)** mandates that any modification of software requires explicit written permission from the copyright holder, reflecting one of the strictest legal stances in Europe [7].
- **Turkey**: According to the **Law No. 5846 on Intellectual and Artistic Works**, Mod APKs are considered derivative works. Their distribution and use without authorization constitute a violation of copyright law [5].
- **Azerbaijan:** National legislation, including the **Law on Copyright and Related Rights**, classifies software as intellectual property. Modifying applications without the author's consent is considered the creation of a derivative work and is legally prohibited. If a modified application is distributed using the original branding, it may also constitute trademark infringement [1].

Despite the existence of legal frameworks, enforcement remains limited. Mod APKs are widely accessible online, and legal action against their distribution is infrequent.

Countermeasures Against Mod APKs

To combat the proliferation of unauthorized applications, major technology platforms have implemented several protective measures:

- **Google Play Store** actively removes applications associated with Mod APKs and may suspend developer accounts involved in such activities.
- **Google Play Protect** scans devices for potentially harmful applications and warns users about security risks.
- **Developers** can issue DMCA takedown notices to request the removal of websites hosting or distributing Mod APKs [11].

These efforts highlight the importance of a collaborative approach involving developers, platform providers, and legal authorities to safeguard the integrity of mobile applications and uphold ethical standards in software distribution.

7. Conclusion and Recommendations

This study has provided a comprehensive analysis of reverse engineering attacks targeting Android mobile applications and the defense mechanisms available to counter them. The open architecture of the Android operating system makes it particularly vulnerable to reverse

engineering, which can lead to the exposure of source code, application logic, and sensitive user data. While tools such as ProGuard and R8 offer basic code obfuscation, they are insufficient on their own. A more effective defense requires a multi-layered approach that includes root detection, anti-debugging techniques, dynamic analysis countermeasures, and the use of native code for critical functionalities [9].

The issue of reverse engineering extends beyond technical boundaries and encompasses significant legal and ethical dimensions. Unauthorized modification of applications, infringement of intellectual property rights, and the leakage of user data are among the most pressing concerns [1], [4], [11]. Therefore, responsibility for application security must be shared not only by developers but also by users and regulatory bodies.

Security is not solely determined by how applications are built, but also by how they are used. Regardless of the strength of implemented protections, careless user behavior can reintroduce vulnerabilities. For this reason, the following recommendations are particularly important for end-users:

- Download applications only from official sources such as the Google Play Store, which conducts security checks to prevent the distribution of malicious software.
- Avoid installing APK files from unknown sources, as they may be modified through reverse engineering or contain malicious code.
- Refrain from rooting devices, as this weakens built-in security mechanisms and increases exposure to attacks.
- Pay close attention to the permissions requested by applications. Unnecessary or suspicious permission requests may indicate potential threats.
- Use reputable antivirus and mobile security software to detect and prevent suspicious activity proactively.

In conclusion, there is no single, definitive solution to securing Android applications. The most effective strategy involves continuously evolving security practices, implementing layered defense mechanisms, and fostering responsible behavior among both developers and users.

Acknowledgment

This research work was carried out by third-year students majoring in Information Technologies at the Azerbaijan State University of Economics (UNEC). We would like to express our sincere gratitude to Mr. Shamil Humbatov, Senior Lecturer at the Department of Digital Technologies and Applied Informatics at UNEC, for his valuable academic guidance and professional support throughout the preparation of this study.

Authors' Declaration

We, the undersigned authors hereby declare that the research article is the result of our original work. All sources of information and data used in the preparation of this study have been properly cited and acknowledged. We affirm that the manuscript has not been submitted to or published in any other journal or platform, either in whole or in part.
We further confirm that all authors have contributed significantly to the conception, development, and finalization of the research, and that there is no conflict of interest related to the content of this work.

Authors' Contribution Statement

- **Shamil Humbatov**: Provided academic supervision and methodological guidance throughout the research process; reviewed and edited the final manuscript.
- **Murad Najafli**: Contributed to the development of the research concept, conducted data analysis, and participated in the writing of the manuscript.
- **Dilbar Guliyeva**: Participated in literature review, data collection, and contributed to the structuring of the research content.

- **Dilbar Amrahova**: Assisted in the preparation of visual materials, formatting of the manuscript, and contributed to the final proofreading and editing.

All authors have read and approved the final version of the manuscript and agree to be accountable for all aspects of the work.

References

1. Azerbaijan Republic, *Law on Copyright and Related Rights*, 1996. [Online]. Available: https://e-qanun.az/framework/4167
2. Caesar, "Clash of Clans MOD APK: The ultimate truth – pros, cons & safe alternatives," *JBMS360*, Mar. 16, 2025. [Online]. Available: https://jbms360.com/clash-of-clans-mod-apk-the-ultimate-truth-pros-cons-safe-alternatives/
3. R. Chandel, "Android Penetration Testing: APK Reverse Engineering," *Hacking Articles*, Feb. 3, 2021. [Online]. Available: https://www.hackingarticles.in/android-penetration-testing-apk-reverse-engineering/
4. European Union, *Directive 2001/29/EC on the harmonisation of certain aspects of copyright and related rights in the information society*, 2001. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A02001L0029-20010622
5. F. Gültekin, "Changes in Law No. 5846 on Intellectual and Artistic Works," 2021. [Online]. Available: https://firatgultekin.com.tr/5846-sayili-fikir-ve-sanat-eserleri-kanunuda-yapilan-degisiklikler/
6. T. Aliyeva, "Investigation of Cyberattack and Intrusion: Methods and Tool", Mastering Intrusion Detection for Cybersecurity [Working Title]. IntechOpen, Feb. 13, 2025. doi: 10.5772/intechopen.1009172.
7. R. Lakshmanan, "Modified WhatsApp App Caught Infecting Android Devices with Malware," *The Hacker News*, Oct. 13, 2022. [Online]. Available: https://thehackernews.com/2022/10/modified-whatsapp-app-caught-infecting.html
8. R. A. Plutte, "Overview of Copyright Protection for Computer Programs in Germany," 2022. [Online]. Available: https://www.ra-plutte.de/ubersicht-zum-urheberrechtsschutz-von-computerprogrammen/
9. A. Smiley, "Android Mobile Reverse Engineering," *Corellium*, 2025. [Online]. Available: https://www.corellium.com/blog/android-mobile-reverse-engineering
10. L. Batyuk et al., "Automated Reverse Engineering of Android Applications," in *Proc. Working Conf. Reverse Engineering (WCRE)*, 2011. [Online]. Available: https://doi.org/10.1109/WCRE.2011.17
11. Y. Shprinz, "Reverse Engineering the Medium App (and Making All Stories in It Free)," *HackerNoon*, Aug. 12, 2018. [Online]. Available: https://medium.com/hackernoon/dont-publish-yet-reverse-engineering-the-medium-app-and-making-all-stories-in-it-free-48c8f2695687
12. U.S. Copyright Office, *The Digital Millennium Copyright Act of 1998*, 1998. [Online]. Available: https://www.copyright.gov/legislation/dmca.pdf