

Vulnerability Research of Mobile Applications Commonly Used in Sweden

How Secure Are Mobile Applications Commonly Used in Sweden Against Vulnerabilities?

Yamini Balannagari

Communication Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

07 January 2025

Supervisor:
Examiner: Guancia Roberto

Abstract

This project explores the implementation and evaluation of an efficient and scalable cryptographic solution for Mobile Crowdsensing (MCS) systems, focusing on addressing key challenges related to security, privacy, and computational efficiency. The study begins with a baseline system using the Elliptic Curve Digital Signature Algorithm (ECDSA) for individual message signing and verification. This initial implementation establishes a foundation for understanding the cryptographic workflows and performance metrics in MCS. Building on this, the project introduces an enhanced architecture employing the Boneh-Lynn-Shacham (BLS) aggregated signature scheme, which allows multiple signatures to be combined into a single compact signature, significantly improving scalability and reducing verification overhead.

The analysis highlights the performance improvements achieved by transitioning from individual verification to aggregated verification, particularly in reducing computational costs as the number of participants grows. The results demonstrate that the BLS aggregated scheme provides substantial efficiency gains while maintaining strong security guarantees. The study concludes by proposing future enhancements, including incorporating anonymity-preserving mechanisms to further protect user privacy in MCS applications.

Contents

1	Introduction	1
1.1	Research Problem	1
1.1.1	Original problem and definition	1
1.1.2	Scientific and engineering issues	2
1.2	Research Question	3
1.3	Purpose	3
1.3.1	Degree Project Purpose	3
1.3.2	Benefits	3
1.3.3	Ethical, Sustainability, and Social Issues	4
1.4	Goals	4
1.5	Delimitations	5
1.6	Structure of the report	5
2	Background	7
2.1	Mobile Application	7
2.1.1	Types of Mobile Applications	8
2.1.2	Latest Statistics on Mobile Applications	8
2.1.3	Components of a Mobile Application	9
2.1.4	Mobile Application Usage in Sweden	9
2.1.5	Key Mobile Application Trends in Sweden	9
2.2	Introduction to Mobile Application Security	10
2.2.1	Importance of Mobile Application Security	10
2.2.2	Common Mobile Application Security Threats	11
2.2.3	Potential Risks in Mobile Applications	11
2.3	Basic Terminology and Concepts	12
2.3.1	Android	12
2.3.2	Android Application (APK)	12
2.3.3	Static vs Dynamic Analysis	12
2.3.3.1	Static Analysis	12
2.3.3.2	Dynamic Analysis	12

2.3.3.3	Calculating the Results for Static and Dynamic Analysis	13
2.3.4	Reverse Engineering	14
2.3.5	Rooted Devices and Emulators	14
2.3.6	Common Exploits and Attack Vectors	15
2.3.7	OWASP Mobile Security Standards	17
2.3.7.1	OWASP Mobile Top 10 Vulnerabilities	18
2.3.7.2	OWASP Mobile Security Testing Guide (MSTG)	19
2.3.7.3	OWASP Mobile Application Security Verification Standard (MASVS)	19
2.3.7.4	Mobile Application Security Checklist	20
2.3.8	Common Exploits and Attack Vectors	22
2.3.9	Testing Methodologies	22
2.4	Android Operating System	24
2.4.1	Key Features of Android	24
2.4.2	Android Architecture Overview	24
2.4.3	Security Model in Android	25
2.4.4	Security Challenges in Android	26
2.5	Related Work	26
2.5.1	Android Architecture and Market Trends	26
2.5.2	Mobile Security Standards and Testing	26
2.5.3	Privacy and Application Security	27
2.5.4	Common Threats and Reverse Engineering	27
3	Methodology	28
3.1	Research Design and Strategy	28
3.1.1	Application Selection Criteria	29
3.2	Experimental Setup	29
3.2.1	Test Environment Configuration	29
3.2.2	Device and OS Specifications	30
3.2.3	Toolchain and Software Stack	31
3.2.4	Tool-Based Workflow and Execution Process	31
3.3	DREAD Threat Modeling	34
3.3.1	Purpose and Application	34
3.3.2	DREAD Components and Scoring Guidelines	34
3.3.3	Scoring Methodology	35
3.3.4	Severity Classification	36
3.3.5	Benefits and Limitations	36
3.4	Risk Prioritization	36

4	Results	38
4.1	Checklist Summary	39
4.2	Checklist Summary and Security Assessment Results per Application	40
4.3	DREAD Scoring Application and Results	43
4.4	DREAD Risk Assessment	44
4.5	Detailed Results and Impact Analysis	45
4.5.1	Foodora	45
4.5.1.1	Sensitive Payment Card Information Stored in Plaintext	45
4.5.1.2	SQL Injection Vulnerabilities in the Cart System	46
4.5.1.3	SSL Pinning and Modification Challenges . . .	47
4.6	Conclusion	48
4.7	Future work	49
4.8	Sustainable development	50
4.9	Ethical considerations	50
	Bibliography	52

List of Figures

2.1	Flowchart of APK Reverse Engineering Process	15
2.2	OWASP Mobile Top 10 Security Risks [1].	17
2.3	Android Operating System Architecture Overview [2].	25
3.1	Tool-Based Security Analysis Workflow	33
4.1	Sensitive Payment Card Information Stored in Plaintext in Foodora	45
4.2	SQL Injection Vulnerability in Foodora’s Cart System	47

List of Tables

2.1	Common Mobile Application Security Threats and their Impacts [1].	11
2.2	Comparison of Static and Dynamic Analysis Techniques	13
2.3	Rooted Devices vs Emulators	14
2.4	OWASP Mobile Top 10 Security Risks [1].	18
2.5	Comparison of Mobile Application Testing Methodologies	23
3.1	Enhanced Profile of Selected Applications and Security-Relevant Features. User engagement data adapted from [3].	30
3.2	Emulator Configuration and OS Specifications	31
3.3	Toolchain and Software Stack Used in the Study	32
3.4	DREAD Risk Classification	36
3.5	Risk Severity Classification	37
4.1	Verified Security Results for Handelsbanken (Based on Static Analysis)	40
4.2	Verified Security Results for Kronans Apotek (Based on Static Analysis)	41
4.3	Verified Security Results for Voi (Based on Static Analysis)	42
4.4	Complete Security Test Results for SL (Based on OWASP MSTG)	42
4.5	Revised Security Test Results for Foodora (Based on OWASP MSTG)	43
4.6	DREAD Score Summary for Mobile Applications	44

List of Acronyms and Abbreviations

OWASP	Open Worldwide Application Security Project
MSTG	Mobile Security Testing Guide
MASVS	Mobile Application Security Verification Standard
APK	Android Application
OS	Operating System
HAL	Hardware Abstraction Layer
ART	Android Runtime
MitM	Man-in-the-Middle
APK	Android Package Kit
AVD	Android Virtual Device
AES	Advanced Encryption Standard

Chapter 1

Introduction

Mobile applications have become an indispensable part of modern life, offering convenience and efficiency in various sectors such as banking, transportation, healthcare, and food delivery. In Sweden, a country recognized as one of the most digitally advanced societies in the European Union, the adoption of mobile applications for essential services has grown significantly [4]. However, this widespread usage has also introduced critical security challenges. Mobile applications often handle sensitive personal, financial, and health data, making them prime targets for cyberattacks. Despite advancements in security measures, vulnerabilities such as weak encryption, poor session management, and insufficient authentication mechanisms continue to pose significant risks to users and organizations alike [5].

This research builds upon previous studies, such as the 2022 evaluation of Swedish mobile applications by Ekenblad and Garrido, which identified persistent vulnerabilities like brute-force attacks and session hijacking [5]. By focusing on five widely used applications in Sweden—Foodora, Handelsbanken, SL, Kronans Apotek, and Voi- e scooter hire. This study aims to identify and address security gaps, ultimately contributing to a safer mobile ecosystem.

1.1 Research Problem

1.1.1 Original problem and definition

The increasing reliance on mobile applications for critical services has exposed users to potential security risks. Vulnerabilities in these applications can lead to unauthorized access, data breaches, and financial fraud, compromising user privacy and trust. Despite the implementation of industry-standard security frameworks, such as the OWASP Mobile Security Testing Guide (MSTG), many

applications still exhibit significant security flaws [6]. These vulnerabilities are often a result of inadequate encryption, weak authentication mechanisms, poor session management, and insufficient API security [7].

1.1.2 Scientific and engineering issues

This research addresses key scientific and engineering challenges relevant to mobile application security. These include:

- **Runtime Vulnerability Detection:** Vulnerabilities such as logic flaws and insecure runtime permissions often appear only during execution. Static analysis alone is insufficient to uncover these flaws, making dynamic analysis essential [8].
- **Reverse Engineering of Obfuscated Applications:** Many production applications use obfuscation and anti-debugging techniques to hinder analysis. Analyzing such apps requires tools like Frida, Jadx, and APKTool to bypass protections and inspect internal logic [5].
- **Session and Authentication Security:** Poor session management, token leakage, and flawed authentication flows are common attack vectors. Identifying these issues involves inspecting session handling, login/logout mechanisms, and token storage [8].
- **Cryptography and Key Management:** Insecure encryption algorithms, hardcoded keys, and improper key storage can compromise confidentiality. This requires assessment of cryptographic implementations and API usage [8].
- **API and Backend Communication Security:** Mobile applications often rely on APIs for backend communication. Testing for insecure endpoints, lack of authorization, or weak transport encryption is essential to identify potential breaches [8].
- **Ethical and Legal Considerations:** Testing real-world applications requires strict adherence to ethical guidelines and legal boundaries. The methodology avoids accessing user data or violating application terms, ensuring responsible and compliant research practices.

1.2 Research Question

The core objective of this research is to assess the real-world effectiveness of security measures implemented in popular mobile applications. Accordingly, the central research question is formulated as follows:

How effective are existing mobile application security mitigation strategies in preventing runtime vulnerabilities, and how can dynamic analysis and reverse engineering be systematically applied to evaluate their adequacy and enhance their effectiveness?

This question seeks to explore the effectiveness of advanced techniques like dynamic analysis and reverse engineering in identifying vulnerabilities that may not be detected through static analysis alone. By focusing on runtime vulnerabilities, the study aims to provide a comprehensive understanding of the security gaps in mobile applications, ultimately contributing to the development of more secure systems [9].

1.3 Purpose

The purpose of this thesis is to systematically identify and analyze security vulnerabilities in five widely used Swedish mobile applications: **Foodora**, **Handelsbanken**, **SL (Storstockholms Lokaltrafik)**, **Kronans Apotek**, and **Voi (e-scooter hire)**. The study employs industry-standard methodologies, including the **OWASP Mobile Security Testing Guide (MSTG)**, to assess the adequacy of implemented security measures. Techniques such as static and dynamic analysis, reverse engineering, and penetration testing are used to uncover potential weaknesses in areas such as authentication, data storage, session management, and **API integration**.

1.3.1 Degree Project Purpose

The broader purpose of the degree project is to demonstrate the ability to apply advanced cybersecurity techniques to real-world applications in a systematic, ethical, and legally compliant manner. It also aims to contribute practical insights to the mobile security domain, reinforcing secure development practices and raising awareness of persistent threats in the mobile ecosystem [6].

1.3.2 Benefits

If the goals of this project are achieved, the following stakeholders will benefit:

- **Users:** Enhanced security of mobile applications will protect users sensitive personal, financial, and health data from unauthorized access and cyberattacks. This will increase user trust in digital services, leading to greater adoption of mobile applications for essential services [7].
- **Developers and Organizations:** Insights from this research will help developers and organizations identify and address security vulnerabilities in their applications, reducing the risk of data breaches and financial losses. This encourages ethical practices by prioritizing user privacy and data security [9].
- **Regulatory Bodies:** The findings of this study can inform policymakers and regulatory bodies about the current state of mobile application security, helping them develop more effective regulations and standards. This promotes the development of sustainable security practices that can be applied across the industry [4].
- **Academic Community:** The research contributes to the academic body of knowledge on mobile application security, providing a foundation for future studies and innovations in the field. This encourages further research and collaboration to address emerging security challenges in the digital age [5].

1.3.3 Ethical, Sustainability, and Social Issues

The research involves identifying vulnerabilities in actively used applications, which raises ethical concerns about potential misuse of the findings. To address this, the study will follow responsible disclosure practices, ensuring that vulnerabilities are reported to the respective application developers and organizations before public disclosure [6].

From a sustainability perspective, promoting secure development practices contributes to the sustainability of digital ecosystems by reducing the environmental and economic costs associated with data breaches and cyberattacks [4].

Socially, the research highlights the importance of security in applications that handle sensitive user data, addressing concerns about privacy and data protection in an increasingly digital world. By raising awareness of these issues, the study aims to foster a culture of security-conscious development and usage [7].

1.4 Goals

The primary goals of this research are:

- To systematically identify and analyze security vulnerabilities in widely used Swedish mobile applications using advanced techniques such as static and dynamic analysis, reverse engineering, and penetration testing.
- To uncover runtime vulnerabilities that may not be detectable through static analysis alone, focusing on areas such as authentication, data storage, session management, and API integration.
- To provide actionable insights and recommendations for developers and organizations to address identified vulnerabilities, thereby enhancing the security of mobile applications and protecting sensitive user data.

1.5 Delimitations

The scope of this thesis project is bounded by the following delimitations, which define what is explicitly included and excluded from the study:

- **Application Selection:** The study focuses on five widely used Swedish mobile applications: **Foodora**, **Handelsbanken**, **SL (Storstockholms Lokaltrafik)**, **Kronans Apotek**, and **Voi (e-scooter hire)**.
- **Platform:** The research is limited to Android-based applications.[10].
- **Time Constraints:** Given the limited timeframe for the research, the study primarily focuses on reverse engineering and dynamic analysis techniques to identify and validate vulnerabilities.
- **Mitigation Strategies:** The study focuses on identifying and analyzing vulnerabilities but does not include the development or evaluation of mitigation strategies. Recommendations for addressing vulnerabilities are provided, but their implementation is outside the scope of this project.

These delimitations ensure the study remains focused and manageable within the given constraints.

1.6 Structure of the report

The report is organized into five major chapters: Introduction, Background, Methodology, Analysis, and Conclusions. The Introduction outlines the project's objectives, delimitations, and research questions, providing a clear starting point for understanding the scope and purpose of the study. The Background

chapter covers the necessary technical concepts and includes a literature review, offering context and insight into related work in the field. Following this, the Methodology is divided into two parts: the baseline implementation and the enhanced implementation, detailing the approaches used to achieve the project goals. The Analysis section assesses the performance of both implementations, providing a thorough evaluation of the results. Finally, the Conclusions summarize the key findings and implications of the study, closing with reflections on the outcomes and potential future directions.

Chapter 2

Background

This chapter provides a comprehensive background on mobile application security, with a focus on the Android ecosystem. It explores the importance of securing mobile apps, especially considering the sensitive user data they handle and their widespread usage across various sectors. The chapter begins by introducing the Android operating system, providing insights into its architecture and security model, which serve as the foundation for understanding potential vulnerabilities in mobile applications.

Next, the chapter discusses common vulnerabilities that affect mobile applications, particularly Android-based apps, and highlights the security risks associated with them. It covers topics such as insecure data storage, weak authentication mechanisms, and exposure to unauthorized access through unprotected communication channels.

The chapter also presents key security standards from OWASP, including the Mobile Top 10 Vulnerabilities, the Mobile Security Testing Guide (MSTG), and the Mobile Application Security Verification Standard (MASVS), which are widely used to assess and address security issues in mobile applications.

Finally, the chapter reviews previous academic studies and industry work on mobile security, identifying research gaps that this thesis aims to address.

2.1 Mobile Application

A mobile application (mobile app) is a software program specifically designed to run on mobile devices such as smartphones, tablets, and smartwatches. These applications leverage device capabilities including cameras, GPS, and sensors to deliver targeted functionality to users.

2.1.1 Types of Mobile Applications

Mobile applications can be categorized based on their functionalities, including:

- **Communication:** Apps like WhatsApp, Facebook Messenger, and Skype, which allow users to send messages, make voice or video calls, and share multimedia.
- **E-Commerce:** Apps for online shopping, such as Amazon, eBay, and Foodora, which allow users to browse products, make purchases, and track orders.
- **Banking and Finance:** Apps for managing financial tasks, such as mobile banking (e.g., Handelsbanken) or personal finance management apps.
- **Healthcare:** Apps like Fitbit or those related to health insurance (e.g., Kronans Apotek), which track fitness data or help users manage health records.
- **Entertainment:** Apps for streaming video or music, such as Netflix, YouTube, and Spotify.
- **E-Mobility:** Voi, an e-mobility scooter app that allows users to rent electric scooters for short trips.
- **Transportation:** SL app, which manages public transportation services in Stockholm, including bus, subway, and train schedules.

2.1.2 Latest Statistics on Mobile Applications

- **Global Smartphone Users:** As of January 2025, there are approximately 5.78 billion unique mobile users worldwide, representing about 70.5% of the global population [11].
- **Mobile App Downloads:** In 2024, global app downloads reached 137.8 billion, a slight decrease of 1% compared to the previous year [3].
- **App Store Offerings:** The Apple App Store hosts around 1.96 million apps, while the Google Play Store offers approximately 2.87 million apps for download [12].
- **User Engagement:** On average, smartphone owners use 10 apps per day and engage with 30 apps per month [12].

- **Revenue Generation:** Mobile apps generated over \$935 billion in revenue in 2024 [12].

These statistics illustrate the pervasiveness and economic impact of mobile applications across the globe.

2.1.3 Components of a Mobile Application

Mobile apps consist of several core components:

- **Code:** The logic of the app, typically written in programming languages such as Java or Kotlin (for Android) or Swift (for iOS). It defines how the app behaves and interacts with the system.
- **Resources:** These include images, layouts, audio, and other assets used within the app. These resources shape the user interface and experience.
- **Manifest:** An XML file (for Android) or a property list (for iOS) that contains information about the app, such as its name, permissions, and settings.

2.1.4 Mobile Application Usage in Sweden

Sweden is among the world's most digitally mature societies, characterized by a high degree of smartphone penetration and extensive reliance on mobile applications for everyday activities. As of 2025, smartphone penetration in Sweden exceeds 90%, positioning mobile apps as fundamental tools in the daily lives of the Swedish population [11].

The widespread adoption of mobile applications in Sweden has significantly impacted the country's social and economic landscapes, influencing consumer behavior, digital payment preferences, public transportation efficiency, healthcare accessibility, and sustainability initiatives. According to recent studies, approximately 60% of the Swedish population regularly downloads and actively uses new mobile applications, reflecting their strong engagement with digital services [3].

2.1.5 Key Mobile Application Trends in Sweden

The following trends highlight the growing importance of mobile applications across various sectors in Sweden:

- **Digital Payments:** Apps like **Swish** are standard in Sweden, used by nearly 70% of the population for everyday transactions.

- **Food Delivery:** Services like **Foodora** have rapidly grown due to consumer preferences for convenience and efficiency.
- **Public Transportation:** The **SL app** is essential for Stockholm commuters, providing real-time updates to over 1.5 million users.
- **Healthcare Access:** Pharmacy apps such as **Kronans Apotek** simplify prescription and medication management digitally.
- **E-Mobility:** Sustainable travel apps like **Voi** are increasingly popular in Swedish cities for short-distance commuting.
- **Mobile Banking:** Banking apps like **Handelsbanken** are widely adopted, offering secure, convenient management of personal finances.

2.2 Introduction to Mobile Application Security

With the exponential growth in mobile application usage, security has emerged as a critical concern, primarily due to the sensitive nature of user data these applications handle. Mobile apps often process personal, financial, healthcare, and location-based information, making them prime targets for malicious actors. Consequently, robust security mechanisms are vital to protect user data, maintain trust, and comply with regulatory requirements like the General Data Protection Regulation (GDPR) [13].

2.2.1 Importance of Mobile Application Security

The significance of mobile application security is underscored by several key factors:

- **Data Sensitivity:** Mobile apps regularly store and manage sensitive personal, financial, and medical data.
- **Widespread Adoption:** With billions of global users, vulnerabilities in mobile apps can have widespread consequences.
- **Regulatory Compliance:** Non-compliance with data protection regulations (e.g., GDPR) can lead to significant legal and financial penalties.
- **User Trust:** Security breaches severely damage user confidence and the organization's reputation.

2.2.2 Common Mobile Application Security Threats

Mobile applications face numerous threats that can compromise data confidentiality, integrity, and availability. Table 2.1 summarizes the common risks and their potential impacts.

Security Threat	Impact
Insecure Data Storage	Unauthorized data access, privacy violations
Weak Authentication	Unauthorized account access, identity theft
Unsecured Communication	Data interception, Man-in-the-Middle attacks
Poor Session Management	Session hijacking, compromised user accounts
Code Tampering	Malicious code injection, compromised functionality
Reverse Engineering	Disclosure of sensitive logic and vulnerabilities

Table 2.1: Common Mobile Application Security Threats and their Impacts [1].

2.2.3 Potential Risks in Mobile Applications

Mobile applications face several potential risks which can compromise user privacy and device security:

- **Data Leakage:** Sensitive information can unintentionally be exposed through insecure storage, unencrypted data transmission, or inadequate permission management.
- **Insecure Communication:** Applications may communicate with servers without proper encryption or validation, leaving data vulnerable to interception and manipulation.
- **Unauthorized Access:** Weak authentication methods and session management flaws can enable unauthorized users to access sensitive information or functionalities within the app.
- **Malicious Code Injection:** Inadequately validated input fields can allow attackers to inject malicious scripts or code, compromising app integrity and user data.
- **Improper Use of Permissions:** Excessive or improperly managed app permissions can expose users to privacy invasions and unintended data access.

2.3 Basic Terminology and Concepts

This section introduces fundamental terms and concepts crucial for understanding mobile application security.

2.3.1 Android

Android is an open-source mobile operating system developed by Google, primarily used for smartphones and tablets. Android is the most widely used mobile operating system globally, with a market share of over 70% due to its open-source nature, versatility, and extensive developer support.

2.3.2 Android Application (APK)

An Android application is packaged and distributed as an Android Package Kit (APK). An APK file typically contains the application's executable code, resources (such as images and layout files), and a manifest file (`AndroidManifest.xml`), which provides crucial information about permissions and configurations required for the app.

2.3.3 Static vs Dynamic Analysis

Static and dynamic analysis are crucial methods employed in mobile application security testing, each offering unique insights into an app's vulnerabilities.

2.3.3.1 Static Analysis

Static Analysis involves examining the application's source code or binaries without executing the program. It is primarily used to detect vulnerabilities such as insecure coding practices, hardcoded credentials, or improper data handling before the application is run. Static analysis helps identify potential security flaws early in the development process, enabling developers to address issues before runtime.

2.3.3.2 Dynamic Analysis

Dynamic Analysis, in contrast, analyzes the application during runtime. It involves monitoring the app's behavior as it operates, identifying vulnerabilities that only manifest when the app is executed, such as runtime errors, insecure data transmission, or improper handling of external services. Dynamic analysis is particularly useful for detecting issues that cannot be observed through static

Criteria	Static Analysis	Dynamic Analysis
Execution	No execution required	Requires application execution
Testing Focus	Code structure, syntax, data flow	Runtime behavior, interactions
Vulnerability Detection	Coding errors, embedded secrets	Runtime vulnerabilities, memory leaks
Analysis Speed	Faster, automated scans	Slower, detailed runtime monitoring
Accuracy	May generate false positives	Fewer false positives, practical insights
Detection Coverage	Broad (code-based issues)	Specific (execution-based issues)
Complexity	Lower complexity	Higher complexity

Table 2.2: Comparison of Static and Dynamic Analysis Techniques

code inspection alone. In practice, effective security testing typically involves a combination of static and dynamic analysis methods, providing comprehensive vulnerability detection throughout the development lifecycle.

2.3.3.3 Calculating the Results for Static and Dynamic Analysis

The results for static and dynamic analysis were calculated based on the following steps:

1. Static Analysis:

- Tools used: MobSF and JADX for APK decompilation and code analysis
- Inspected elements:
 - Hardcoded credentials (API keys, database passwords)
 - Insecure data storage (plaintext sensitive data)
 - Improper permission handling in AndroidManifest.xml
- Evaluation criteria:
 - **Pass**: Adherence to best practices (secure storage, proper permissions)
 - **Fail**: Presence of vulnerabilities (insecure storage, hardcoded credentials)
 - N/A: Test not applicable (e.g., no personal data handling)

2. Dynamic Analysis:

- Tools used: Frida, Objection, Burp Suite for runtime observation
- Checked runtime behaviors:
 - Insecure communication (unencrypted transmissions)
 - Authentication failures (weak token/session management)
 - Memory leaks and race conditions
- Evaluation criteria:
 - **Pass**: Secure communication (HTTPS), proper session management
 - **Fail**: Insecure data transmission, authentication flaws
 - N/A: Test not applicable (e.g., no API calls requiring encryption)

This methodology combines static code examination with runtime behavior analysis to provide a comprehensive security assessment. Each test case is scored based on severity and impact, with results prioritized using the DREAD risk scoring model for subsequent remediation planning.

2.3.4 Reverse Engineering

Reverse engineering is the process of analyzing compiled application files (like APKs) to understand their internal logic, functionality, and potential security issues. Security researchers often apply reverse engineering to detect hidden vulnerabilities, logic flaws, or malicious code embedded within applications [14].

2.3.5 Rooted Devices and Emulators

Rooted devices are Android devices modified to grant full administrative (root) access, bypassing built-in security restrictions. They facilitate deep security analysis by providing unrestricted access to the system.

Emulators are software-based tools that mimic Android devices on desktop computers. They offer safe, controlled environments to test applications without needing physical devices, thus simplifying testing processes [15].

Table 2.3 compares these concepts briefly:

Concept	Purpose in Security Testing
Rooted Devices	Allow unrestricted access for deep vulnerability analysis
Emulators	Provide virtual, risk-free environments for testing applications

Table 2.3: Rooted Devices vs Emulators

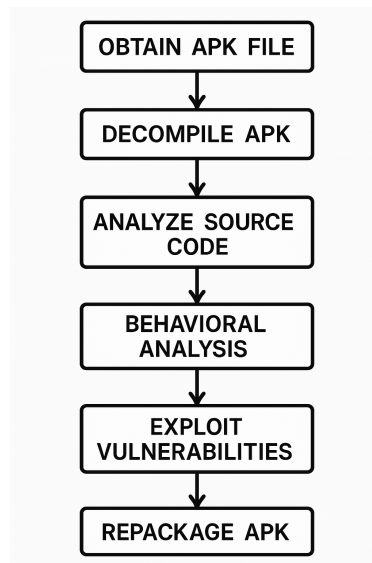


Figure 2.1: Flowchart of APK Reverse Engineering Process

2.3.6 Common Exploits and Attack Vectors

Reverse engineering and dynamic analysis have already been discussed in earlier sections as critical techniques that, while valuable for legitimate security testing, are often exploited by malicious actors. Reverse engineering enables attackers to decompile APK files and extract sensitive information such as hardcoded credentials, encryption keys, or internal API structures [14]. This understanding is then used to identify weaknesses or repackage the application for malicious purposes. Similarly, dynamic analysis allows attackers to observe and manipulate an application's behavior at runtime—particularly when executed on rooted devices or emulators [15]. Through dynamic instrumentation, they can bypass authentication mechanisms, modify memory, and intercept sensitive transactions.

In addition to these techniques, mobile applications face numerous other sophisticated attack vectors:

- **Man-in-the-Middle (MitM) Attacks:** This type of attack occurs when an adversary intercepts data transmitted between a mobile application and its backend server. If the app does not enforce strict TLS protocols or fails to validate server certificates correctly, attackers can view, modify, or inject malicious data into the communication stream [1, 15]. Such attacks are particularly dangerous in public Wi-Fi environments, where network traffic can be easily monitored.
- **Code and Injection Attacks:** Injection vulnerabilities occur when untrusted

user input is improperly handled, allowing attackers to execute unintended commands or queries. For example, SQL injection may allow attackers to read or manipulate backend databases, while JavaScript or shell injection could compromise the app's logic or data. In mobile apps, such attacks often exploit weak input validation in forms, URL parameters, or inter-process communication. These flaws can be particularly dangerous when combined with insecure backend APIs or logging mechanisms [1, 15].

- **Insecure Data Storage:** Mobile apps often store data locally on the device for caching, offline access, or user convenience. However, if sensitive information such as access tokens, credentials, or personally identifiable information (PII) is stored in plaintext without encryption, it can be easily extracted from local databases, shared preferences, or log files [16, 7]—especially on rooted or compromised devices.
- **Excessive Permission Usage:** Applications sometimes request permissions that are not essential to their core functionality. This creates unnecessary security exposure by allowing access to sensitive components like the device's microphone, camera, contacts, SMS, or location data [1, 5]. If such permissions are granted, malware or compromised components within the app may abuse them to collect private user data or conduct surveillance without user consent.
- **Session Hijacking:** Many mobile applications use tokens to maintain user sessions. If these tokens are not securely generated, transmitted, or stored, attackers may intercept or guess them—gaining unauthorized access to the user's account [15, 9]. In the absence of mechanisms like token expiration, regeneration upon logout, or device binding, attackers can maintain persistent access.
- **Application Repackaging:** A common technique in Android threats involves downloading a legitimate app, modifying its code to include malicious payloads, and redistributing it as a trojan. These repackaged apps often appear visually identical to the original and may continue functioning as expected, all while executing unauthorized background operations [15, 1].
- **Overlay and Phishing Attacks:** Attackers can abuse Android's draw-over-permission or accessibility services to display fake login screens on top of legitimate applications. When users unknowingly enter their credentials into these overlays, the information is captured and sent to remote servers [1]. These attacks are particularly effective against banking and financial apps.

- **Improper Error Handling and Logging:** Developers often log system messages, API responses, or exception details for debugging purposes. If these logs are not properly sanitized or are accessible in production builds, they can expose internal workings of the app, including stack traces, endpoint URLs, authentication flows, or even partial user data—providing attackers with a blueprint for exploitation [15].

Collectively, these attack vectors demonstrate the multifaceted nature of mobile application threats. Each vector targets a different aspect of the application lifecycle, from development and deployment to end-user interaction. Therefore, it is imperative to adopt a holistic security strategy that addresses all potential points of compromise.

2.3.7 OWASP Mobile Security Standards

The Open Web Application Security Project (OWASP) is a globally recognized nonprofit organization that provides free and open resources for mobile application security. OWASP's mission is to raise awareness about the importance of security in the development of software and applications. In the context of mobile applications, OWASP has created several critical frameworks and standards that serve as a guide for developers and security testers to identify vulnerabilities and secure mobile applications.



Figure 2.2: OWASP Mobile Top 10 Security Risks [1].

2.3.7.1 OWASP Mobile Top 10 Vulnerabilities

One of OWASP's most important contributions to mobile app security is the **OWASP Mobile Top 10**. This list highlights the top ten most critical vulnerabilities commonly found in mobile applications. It provides a clear framework for developers and security teams to address potential weaknesses that can compromise the confidentiality, integrity, and availability of mobile applications. The vulnerabilities listed in the OWASP Mobile Top 10 are updated regularly to reflect emerging threats and trends in the mobile ecosystem.

The OWASP Mobile Top 10 vulnerabilities are as follows:

Vulnerability	Description
M1: Improper Platform Usage	Misuse of platform features or failure to follow platform-specific security best practices.
M2: Insecure Data Storage	Storing sensitive data (e.g., passwords, credit card details) insecurely, such as in plain text.
M3: Insecure Communication	Failing to use secure communication protocols (e.g., HTTPS) for transmitting sensitive data.
M4: Insecure Authentication	Weak or inadequate authentication mechanisms, including weak passwords or lack of multi-factor authentication.
M5: Insufficient Cryptography	Weak or absent cryptography for encrypting sensitive data, making it vulnerable to exposure.
M6: Insecure Authorization	Issues with authorization processes, where users can access resources they shouldn't be able to.
M7: Client Code Quality	Poor client-side code that could be reverse-engineered to exploit vulnerabilities or reveal secrets.
M8: Code Tampering	Modifying or tampering with the app's code to bypass security controls or exploit vulnerabilities.
M9: Reverse Engineering	Analyzing the app's code to discover weaknesses or secrets that could be exploited.
M10: Extraneous Functionality	Leaving unused or unnecessary functions in the app that could be exploited.

Table 2.4: OWASP Mobile Top 10 Security Risks [1].

The OWASP Mobile Top 10 serves as a comprehensive reference for mobile application security, providing a structured approach for identifying, addressing, and mitigating the most critical risks in mobile apps.

2.3.7.2 OWASP Mobile Security Testing Guide (MSTG)

The **OWASP Mobile Security Testing Guide (MSTG)** is another critical resource developed by OWASP to help mobile app developers and testers ensure that their applications are secure. MSTG provides detailed guidelines, methodologies, and best practices for performing security testing on mobile applications. The guide covers both Android and iOS platforms and includes a comprehensive set of tests for security vulnerabilities, from static code analysis to dynamic testing.

Key features of MSTG include:

- **Comprehensive Test Cases:** MSTG provides detailed test cases and methodologies for assessing various aspects of mobile app security, including authentication, cryptography, data storage, and secure communication.
- **Security Testing Checklist:** A well-organized security checklist that testers can use to ensure that all critical security aspects of the mobile application are tested and validated.
- **Testing Methodologies:** MSTG explains both static and dynamic testing methods to identify vulnerabilities, including penetration testing and code reviews.
- **Platform-Specific Testing:** It covers security testing for both Android and iOS applications, addressing platform-specific security concerns and potential attack vectors.

The MSTG is widely used in the industry and is considered a fundamental resource for any security professional testing mobile applications.

2.3.7.3 OWASP Mobile Application Security Verification Standard (MASVS)

The **OWASP Mobile Application Security Verification Standard (MASVS)** is another essential resource aimed at providing a set of security requirements for mobile applications. MASVS is designed to be used as a framework for evaluating whether a mobile app meets the necessary security standards and guidelines. It helps developers and testers verify that a mobile app adheres to best practices and is secure enough to protect user data and prevent security breaches.

The MASVS is divided into multiple levels:

- **MASVS Level 1:** Focuses on basic security requirements that all mobile apps should meet, regardless of their purpose.

- **MASVS Level 2:** Provides a more advanced set of security requirements, typically applicable to apps that process sensitive data or perform critical tasks.
- **MASVS Level 3:** Represents the highest level of security requirements, intended for mobile applications that require top-tier security measures, such as financial or government applications.

MASVS helps guide the development and testing processes by providing clear criteria for evaluating an app's security posture. It can be used to ensure compliance with industry standards and regulatory requirements.

2.3.7.4 Mobile Application Security Checklist

Ensuring the security and resilience of mobile applications against evolving threats necessitates a methodical and standards-driven approach throughout the development lifecycle. The following checklist, informed by the OWASP Mobile Security Testing Guide (MSTG) [15, 8] and the Mobile Application Security Verification Standard (MASVS) [16], delineates critical security domains that developers and security practitioners should assess when designing and implementing secure mobile applications.

- **Data Storage Security**
 - Sensitive data should only be stored on the device when strictly necessary, and with adequate protection.
 - Secure storage mechanisms, such as the Android Keystore or iOS Keychain, must be used to safeguard confidential data.
 - Storing authentication tokens, credentials, or personal information in plaintext must be strictly avoided.
- **Secure Communication**
 - All data exchanged between the application and backend servers must be transmitted over secure channels using TLS (HTTPS).
 - SSL certificates should be properly validated; insecure configurations such as accepting all certificates must be avoided.
 - API keys, secrets, or session tokens must not be hardcoded within the application binary.
- **Authentication and Session Management**

- Applications should enforce robust user authentication, such as OAuth 2.0, to prevent unauthorized access.
- Secure session tokens must be implemented and appropriately expired or revoked during logout or timeout events.
- Authentication data must not be stored in unprotected or insecure local storage environments.
- **Code Security**
 - Source code should be obfuscated to deter reverse engineering and unauthorized access to internal logic [14].
 - Debugging information, verbose logging, and unused permissions must be removed prior to release.
 - Integrity checks such as checksums or digital signatures should be utilized to detect application tampering.
- **Input Validation**
 - Comprehensive validation should be enforced for all client and server-side inputs to mitigate injection attacks.
 - Output data must be sanitized and encoded to prevent cross-site scripting (XSS) and related vulnerabilities.
- **Permission Handling**
 - Applications must adhere to the principle of least privilege, requesting only the permissions essential to core functionality [1].
 - Clear justifications for permission requests should be provided to users, and their decisions should be handled appropriately.
- **Logging and Debugging**
 - Logging mechanisms must not capture or expose sensitive user data, such as passwords or financial information.
 - Logs should be securely stored and access-controlled to prevent misuse by unauthorized entities.

This structured checklist serves as a foundational guideline for secure mobile application development. Adhering to these practices not only minimizes security vulnerabilities but also ensures compliance with established industry standards and data protection regulations.

2.3.8 Common Exploits and Attack Vectors

Mobile applications are susceptible to a variety of exploits and attack vectors due to their widespread usage, diverse ecosystems, and often limited user awareness. These vulnerabilities are commonly leveraged by attackers to compromise application data, user privacy, or system integrity. Understanding these vectors is essential for designing effective defenses.

- **Reverse Engineering:** Attackers use tools like JADX or Apktool to decompile APK files and extract sensitive logic, hardcoded credentials, or proprietary algorithms [14].
- **Man-in-the-Middle (MitM) Attacks:** Insecure communication channels allow adversaries to intercept, read, and modify data transmitted between the app and its server if TLS is not properly implemented.
- **Code Injection:** Malicious actors exploit poorly validated input fields to inject executable scripts or SQL commands, leading to unauthorized control or data manipulation.
- **Insecure Storage Exploits:** Sensitive information stored in plaintext (e.g., SharedPreferences or local databases) can be easily extracted from rooted devices or through backup analysis.
- **Privilege Escalation:** Attackers may exploit flaws in permission handling or sandboxing mechanisms to gain elevated access and manipulate protected data or system resources.
- **Repackaging and Trojans:** Legitimate applications can be repackaged with malicious payloads and redistributed through unofficial app stores, compromising end users.
- **Session Hijacking:** Weak or improperly managed session tokens can be intercepted or predicted, allowing attackers to impersonate users without valid credentials.
- **Keylogging and Screen Overlay Attacks:** Malicious apps running in the background may log user input or overlay fake login screens to steal credentials, especially on rooted devices.

2.3.9 Testing Methodologies

In cybersecurity and software quality assurance, testing methodologies define how applications are examined for vulnerabilities based on the tester's knowledge of the internal workings of the system.

Table 2.5: Comparison of Mobile Application Testing Methodologies

Criteria	Black-Box Testing	White-Box Testing	Grey-Box Testing
Definition	External testing without access to source code or internal details	Internal testing with full access to source code	Partial access to internal structure and code
Access to Source Code	Not available	Fully available	Partially available
Knowledge of System Internals	None	Complete	Limited
Primary Focus	Authentication, input validation, error handling	Logic flaws, code paths, configuration issues	Session control, API security, runtime analysis
Tool Examples	Burp Suite, MobSF (API scan mode)	SonarQube, Fortify, manual review	Frida, Objection, MobSF (hybrid mode)
Realism of Threat Simulation	High	Low	Medium

For mobile applications, the most widely recognized approaches are **black-box**, **white-box**, and **grey-box** testing. Each of these techniques varies in depth, visibility, and realism when simulating real-world attacks. **Black-box testing** is an external testing method where the tester has no knowledge of the internal codebase or system architecture. The test is conducted from the perspective of an end-user or external attacker, focusing only on the application's inputs and outputs.

White-box testing, in contrast, is a fully transparent approach where the tester has complete access to the source code, architecture, and documentation. This method is useful for deep static analysis, logic verification, and code review.

Grey-box testing combines elements of both black-box and white-box testing. The tester has limited internal knowledge, such as API documentation, decompiled code, or test credentials. This method provides a balanced view by simulating real-world conditions while retaining enough access to discover complex flaws.

In this study, grey-box testing was employed to enable a realistic evaluation of mobile application vulnerabilities while leveraging partial internal access. This approach allowed dynamic manipulation, reverse engineering, and access control testing in controlled environments.

2.4 Android Operating System

The Android Operating System (OS) is a Linux-based, open-source platform primarily used for smartphones, tablets, wearables, and other connected devices. Developed by Google and maintained by the Open Handset Alliance, Android powers over 70% of the world's smartphones as of 2025, making it the most widely used mobile OS globally. Its open-source nature and flexibility have led to its widespread adoption across multiple device types and usage contexts.

2.4.1 Key Features of Android

Android has several key advantages that distinguish it from other mobile operating systems:

- **Open-Source Platform:** Android is based on the Linux kernel and is open-source, allowing anyone to view, modify, and enhance the code, encouraging innovation and rapid development.
- **Customizability:** Android permits users and manufacturers to customize interfaces, install third-party apps, and alter core system features, offering highly personalized user experiences.
- **Multi-platform Support:** Beyond smartphones, Android supports devices such as smartwatches (Wear OS), televisions (Android TV), and automotive systems (Android Auto), contributing to its extensive adoption.

2.4.2 Android Architecture Overview

Android's architecture comprises distinct layers working collaboratively to deliver comprehensive functionality, as illustrated in Figure 3.1:

- **Linux Kernel:** Forms the foundation of Android, managing memory, processes, drivers, and hardware interaction.
- **Hardware Abstraction Layer (HAL):** Acts as an interface between device hardware and software components, allowing app access to hardware functionalities like audio, camera, and sensors.
- **Native Libraries and Android Runtime:** Includes native libraries essential for graphics rendering, multimedia processing, and database management. The Android Runtime (ART) compiles app bytecode into optimized machine code.

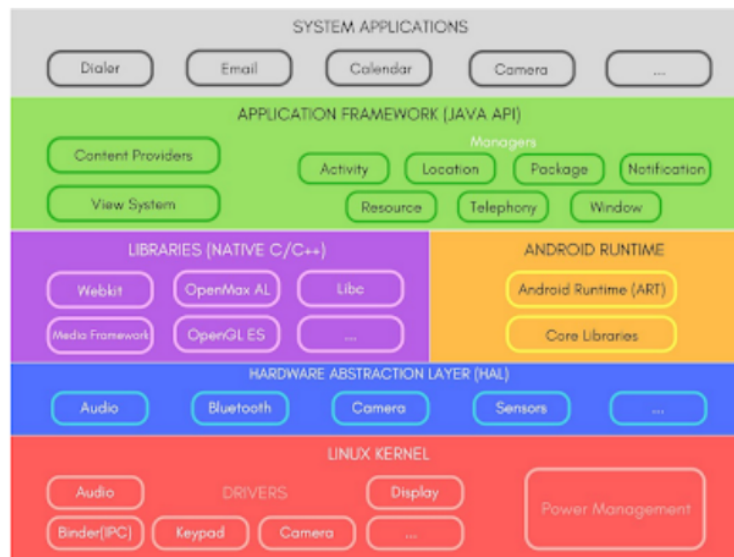


Figure 2.3: Android Operating System Architecture Overview [2].

- **Application Framework (Java API):** Provides developers with APIs for utilizing system services such as notifications, resource management, location services, telephony, and activity management.
- **System Applications:** Comprises built-in Android applications such as the Dialer, Email, Calendar, and Camera, leveraging the Application Framework for functionality.

2.4.3 Security Model in Android

Android utilizes a multi-layered security model to protect user data, prevent unauthorized access, and ensure OS and app integrity, incorporating features such as:

- **App Sandbox:** Each app operates in isolation, safeguarding against unauthorized data access between apps.
- **Permissions:** Explicit user permissions are required for apps to access sensitive device resources like contacts, cameras, and location.
- **App Signing:** Digital certificates validate app authenticity and integrity, preventing unauthorized modifications.

- **Encryption:** Supports robust encryption at both file and application levels, enhancing data security.
- **Google Play Protect:** Regularly scans installed apps for malware and security threats, maintaining device protection.

2.4.4 Security Challenges in Android

Despite a robust security framework, Android faces certain challenges:

- **Fragmentation:** Diverse device manufacturers and Android versions result in delayed security updates, leaving some devices vulnerable to known threats.
- **Malware and Malicious Apps:** Sideloading applications from unofficial sources increases the risk of malware, even with Google Play Protect.
- **Rooting Risks:** Rooting grants administrative access, circumventing security mechanisms and potentially exposing devices to vulnerabilities.

2.5 Related Work

Research in the area of mobile operating systems and mobile application security has been extensive. Various works have explored Android's architecture, market penetration, security mechanisms, and the challenges associated with mobile application development and deployment.

2.5.1 Android Architecture and Market Trends

Google's Android operating system has been extensively documented in official resources [17, 18]. It dominates the mobile OS market globally, with reports indicating a consistent market share of over 70% [10]. The architecture of Android is layered, promoting modularity and security, as elaborated in various academic and technical resources [2].

2.5.2 Mobile Security Standards and Testing

Security testing for mobile applications is well documented by OWASP through the Mobile Security Testing Guide (MSTG) and the Mobile Application Security Verification Standard (MASVS) [8, 15, 16]. These resources provide developers with checklists and methodologies for testing mobile apps against known vulnerabilities.

2.5.3 Privacy and Application Security

Studies such as those by Stirparo (2015) have explored privacy concerns related to personal data in mobile applications [7]. Ekenblad and Garrido (2022) evaluated ten Swedish apps and discovered recurring issues like excessive permission requests and unencrypted data transmissions [5]. Nilsson (2020) explored penetration testing strategies for Android applications, identifying gaps in security practices [9].

2.5.4 Common Threats and Reverse Engineering

OWASP's Mobile Top 10 highlights the most pressing risks in mobile applications, including insecure communication, improper session handling, and reverse engineering vulnerabilities [1]. The threat posed by reverse engineering of APKs has also been addressed in scholarly journals [14].

Chapter 3

Methodology

This chapter outlines the methodology adopted to systematically evaluate the security of Android mobile applications. The approach combines theoretical guidelines from established frameworks—such as the OWASP Mobile Security Testing Guide (MSTG) [15] and the Mobile Application Security Verification Standard (MASVS) [16]—with practical, hands-on testing using a range of automated and manual tools.

The methodology is structured into clearly defined phases: research design, experimental setup, testing procedures, vulnerability assessment, threat modeling, and result reporting. Each phase is designed to simulate real-world attack scenarios under ethical constraints and contribute to a comprehensive analysis of mobile app vulnerabilities.

The study follows a hybrid approach incorporating both static and dynamic analysis to uncover security flaws in application code, runtime behavior, storage, communication, and permissions. Tools such as MobSF, Frida, JADX, apktool, and Burp Suite were selected for their compatibility with Android applications and alignment with OWASP testing practices. A comprehensive overview of these tools and their respective roles in the testing process is provided in Section 3.2.3.

To ensure clarity and reproducibility, the methodology includes visual illustrations, tables, and citations where applicable, enhancing transparency and academic rigor in the vulnerability assessment process.

3.1 Research Design and Strategy

The research adopts a practice-oriented, exploratory approach aimed at analyzing and assessing the security posture of real-world Android applications. The core strategy integrates industry-standard testing methodologies and widely recognized security frameworks to simulate realistic threat scenarios while maintaining

ethical compliance.

The research emphasizes hands-on experimentation using both black-box and gray-box testing approaches. Static and dynamic analysis techniques are applied to evaluate critical security areas such as insecure storage, authentication mechanisms, data transmission, and improper permission handling. These security aspects are examined through the lens of the OWASP Mobile Security Testing Guide (MSTG) [15], which serves as the foundational reference for vulnerability classification and assessment procedures.

This section defines the overarching goals of the study and outlines the rationale for the adopted research design, including the selection of tools, the configuration of the testing environment, and the application of OWASP-aligned testing standards.

3.1.1 Application Selection Criteria

To ensure real-world relevance and contextual diversity, this study selected Android applications spanning several high-impact categories. These include banking, healthcare, food delivery, transportation, and micro-mobility—domains that typically involve sensitive user data and high interaction rates.

Table 3.1 presents an overview of the selected applications, including their functional category, download statistics, user engagement rates, and identified sensitive features.

3.2 Experimental Setup

This section presents the configuration of the experimental environment used for mobile application security testing. The entire process was conducted using a virtualized Android emulator, simulating a realistic application environment while enabling controlled and reproducible testing workflows. The setup supports both static and dynamic testing methodologies, aligned with OWASP MSTG guidelines [15].

3.2.1 Test Environment Configuration

The test environment was built on an Android Virtual Device (AVD) running Android 10 (API 29) using x86 architecture. No physical rooted device or Linux-based host was used. Instead, all analysis tasks were performed using Android Studio's built-in emulator. This emulator was configured to allow certificate installation, proxy redirection, and dynamic instrumentation.

Application Name	Category	Downloads	Engagement (%)	Sensitive Permissions / Features
Foodora	Food Delivery	1M+	50%	Location, Camera, Storage, API Keys in Shared Preferences
Handelsbanken	Banking	1M+	22%	Camera, SMS, Location, Authentication Tokens
SL	Public Transport	1M+	55%	Location, API Traffic Interception, Unsecured Endpoints
Kronans Apotek	Healthcare / Pharmacy	500K+	45%	Storage, Location, Shared Preferences, Insecure Data Storage
Voi	Micro-Mobility	1M+	40%	Location, Bluetooth, Camera, No Root Detection

Table 3.1: Enhanced Profile of Selected Applications and Security-Relevant Features. User engagement data adapted from [3].

- **Platform:** Android Emulator via Android Studio
- **API Level:** 29 (Android 10 "Q")
- **Architecture:** x86
- **Proxy Interception:** Configured via Burp Suite Community Edition
- **Certificate Injection:** Burp CA certificate manually added to emulator store

This virtualized environment allowed both passive and active testing techniques to be conducted safely, including runtime memory inspection, API traffic interception, and static reverse engineering.

3.2.2 Device and OS Specifications

Table 3.2 summarizes the virtual device specifications used for this study, ensuring compatibility with security tools and ease of dynamic manipulation.

Parameter	Specification
Android Version	Android 10.0 ("Q")
API Level	29
Architecture	x86
Google Play Services	Enabled
Emulator Type	Android Virtual Device (AVD) via Android Studio
Network Proxy	Burp Suite (HTTPS/HTTP Interception)
Installed Tools	MobSF, Frida, Objection, JADX, Androbug Scanner
Root Access	Emulated through Frida/Objection
Certificate Pinning Bypass	Frida scripts and Objection hooks

Table 3.2: Emulator Configuration and OS Specifications

3.2.3 Toolchain and Software Stack

The security testing involved a combination of reverse engineering tools, dynamic analyzers, scanners, and traffic interceptors. Table 3.3 presents the list of tools and their roles in the assessment workflow.

Each tool played a complementary role in assessing mobile security from multiple perspectives—code, network, and runtime—following industry best practices and aligned with the OWASP MSTG [15, 8].

3.2.4 Tool-Based Workflow and Execution Process

To execute the security testing methodology, a structured workflow was adopted for applying each tool. The tools were used in a complementary fashion to ensure complete coverage of both static and dynamic security concerns.

- **MobSF (Mobile Security Framework):** Each APK was initially scanned using MobSF to perform automated static analysis. MobSF generated reports on permissions, manifest configuration, code structure, network endpoints, and cryptographic practices. It also highlighted potential vulnerabilities such as hardcoded secrets and exposed components.
- **JADX (Decompilation):** Decompiled APKs were opened in JADX to manually inspect the Java source code. This allowed for deeper exploration of the app's logic, hardcoded credentials, API endpoints, encryption routines, and hidden debug flags. It was also used to trace methods referenced in Frida scripts.

Tool	Purpose and Rationale
MobSF	Comprehensive static and dynamic analysis, including manifest inspection, API checks, and basic malware detection. Used to generate OWASP-aligned reports.
Frida	Dynamic instrumentation framework used for hooking API calls, bypassing security features (e.g., root detection), and altering app behavior at runtime.
Objection	Built on Frida; provides CLI access to runtime controls, local storage, bypass modules, and dynamic interaction with Android internals.
JADX	Decompilation tool used to convert APK bytecode into readable Java code, assisting in identifying hardcoded secrets, business logic, and control flow.
Androbug Scanner	Lightweight vulnerability scanner used to analyze APKs for exposed components, weak coding patterns, and known CVEs.
Burp Suite	Used as a proxy to intercept and analyze all HTTP and HTTPS traffic. CA certificate was injected into the emulator for SSL decryption. Useful for identifying insecure communication and broken certificate validation.
Custom Frida Scripts	Custom injection logic used for modifying runtime values, simulating exploit conditions, and analyzing app response to manipulated APIs.

Table 3.3: Toolchain and Software Stack Used in the Study

- **Burp Suite:** Burp Suite Community Edition was configured as an HTTP/HTTPS proxy to intercept and decrypt network communication. The Burp CA certificate was manually installed on the emulator to enable SSL inspection. App traffic—including login attempts, session tokens, and API payloads—was monitored for sensitive data exposure or insecure transport.
- **Frida:** Frida was used for dynamic binary instrumentation. Custom Frida scripts were developed to hook into specific Java and native functions, modify method return values, bypass security checks (e.g., root/jailbreak detection), and monitor encryption or authentication processes at runtime.
- **Objection:** Serving as a command-line interface over Frida, Objection was used to explore the file system, read SharedPreferences, bypass root and SSL pinning, and dynamically invoke app components. It accelerated real-

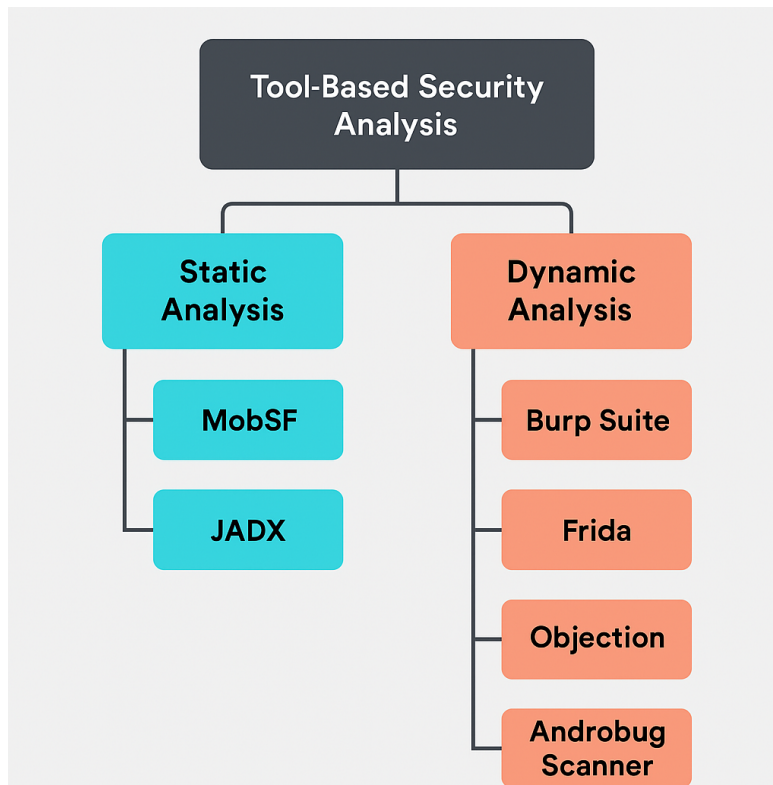


Figure 3.1: Tool-Based Security Analysis Workflow

time interaction with the running app and helped automate common testing routines.

- **Androbug Scanner:** This lightweight scanner supplemented static analysis by highlighting configuration risks such as exported activities, missing permission enforcement, and misconfigured broadcast receivers or services.
- **Custom Frida Scripts:** Tailored scripts were written to automate the inspection of runtime-sensitive functions such as cryptographic APIs, logging methods, and token validators. These scripts provided deeper runtime insights beyond what standard Objection modules offered.

This integrated toolchain was applied consistently across all test cases. Logs, screenshots, and captured traffic were preserved to support vulnerability validation and result reporting.

3.3 DREAD Threat Modeling

Threat modeling is an essential step in secure software design, allowing developers and security analysts to proactively identify and mitigate potential risks. Among the various models available, the DREAD threat model—developed by Microsoft—offers a structured, quantitative framework to evaluate and prioritize security vulnerabilities. The acronym **DREAD** stands for *Damage Potential*, *Reproducibility*, *Exploitability*, *Affected Users*, and *Discoverability*. Each factor helps assess a specific aspect of the risk posed by a threat, and the resulting scores help guide mitigation efforts [19].

3.3.1 Purpose and Application

DREAD is primarily used to quantify the risk associated with individual security threats. By assigning scores to each of the five components, organizations can determine which vulnerabilities pose the greatest risk and require immediate attention. In this study, the DREAD model is used to evaluate vulnerabilities identified in mobile applications through static and dynamic testing. The goal is to provide a consistent method to rank threats in terms of severity and exploitability.

3.3.2 DREAD Components and Scoring Guidelines

Each DREAD category is scored on a scale from 1 (least severe) to 10 (most severe), based on established criteria. The detailed evaluation methodology is as follows:

- **Damage Potential (D):** This refers to the extent of harm that could result if the threat were successfully exploited. It includes potential financial loss, reputational damage, data integrity compromise, or system availability disruption. A higher score is assigned to threats that could significantly impact business operations or user safety.
 - 1–3: Minor inconvenience or UI-related issues.
 - 4–6: Moderate impact on application functionality or partial data exposure.
 - 7–10: Complete data breach, financial fraud, or system takeover.
- **Reproducibility (R):** This measures how consistently the attack can be repeated. Vulnerabilities that work reliably under various environments or do not require specific conditions score higher in this category.

- 1–3: Very difficult to reproduce; requires timing or rare conditions.
 - 4–6: Reproducible with some preparation or user interaction.
 - 7–10: Easily reproducible on any system without special requirements.
- **Exploitability (E):** This indicates the complexity of carrying out the exploit. Lower effort and tool availability result in higher scores. It considers factors such as access level, technical skills, and availability of public exploits or scripts.
 - 1–3: Requires physical access or highly sophisticated tools.
 - 4–6: Needs intermediate skills and partial knowledge of system internals.
 - 7–10: Can be exploited remotely using publicly available tools or minimal technical knowledge.
 - **Affected Users (A):** This evaluates the potential reach of the vulnerability in terms of user base. It helps estimate the scale of impact from a user perspective.
 - 1–3: Affects only a niche or test group.
 - 4–6: Affects a significant subset of users.
 - 7–10: Impacts all or most users of the application or service.
 - **Discoverability (D):** This refers to how easily the vulnerability can be identified by an attacker. Easily discoverable issues score higher because they pose a greater risk of real-world exploitation.
 - 1–3: Deeply hidden; requires extensive reverse engineering or specialized knowledge.
 - 4–6: Potentially visible through casual inspection of logs, API behavior, or traffic.
 - 7–10: Easily found in user interface, static code, or metadata.

3.3.3 Scoring Methodology

The final risk score for a given threat is computed using the average of the five component scores:

$$\text{DREAD Score} = \frac{D + R + E + A + D}{5}$$

This formula provides a single risk metric that can be used for ranking and prioritization.

3.3.4 Severity Classification

The resulting DREAD score is categorized into one of four severity levels:

Table 3.4: DREAD Risk Classification

Score Range	Severity Level
8.1 – 10.0	Critical
6.1 – 8.0	High
3.1 – 6.0	Medium
1.0 – 3.0	Low

This classification aids in identifying which vulnerabilities require immediate remediation and which can be scheduled for later fixes based on available resources.

3.3.5 Benefits and Limitations

The DREAD model offers a number of advantages:

- Provides a quantitative and repeatable method for ranking vulnerabilities.
- Facilitates structured comparison between different threat scenarios.
- Encourages collaborative threat assessment with clearly defined parameters.

However, it also has certain limitations:

- Scoring can be subjective without standardized guidelines.
- Some complex threats may not be adequately captured by simple numerical values.

Despite these limitations, DREAD remains a valuable tool in academic research and experimental threat modeling, especially for its simplicity and ease of integration into security assessment workflows [20, 21].

3.4 Risk Prioritization

After identifying security vulnerabilities through automated and manual testing, risk prioritization was carried out to determine which issues warranted immediate remediation. Each vulnerability was evaluated on two core dimensions:

- **Exploitability:** The ease with which an attacker can exploit the vulnerability, considering technical complexity, access level required, and tool availability.
- **Impact:** The potential consequences of a successful exploit, including data theft, system compromise, or user impersonation.

Each vulnerability was assigned a score from 1 (Low) to 10 (Critical) in both categories. The final risk score was computed using the following formula:

$$\text{Risk Score} = \frac{\text{Exploitability} + \text{Impact}}{2}$$

Based on the computed score, vulnerabilities were classified into severity levels for prioritization.

Table 3.5: Risk Severity Classification

Score Range	Risk Level
8.1 – 10.0	Critical
6.1 – 8.0	High
3.1 – 6.0	Medium
1.0 – 3.0	Low

This classification allowed the creation of a remediation roadmap, prioritizing fixes that addressed the most damaging and exploitable issues first.

Chapter 4

Results

This chapter presents the results obtained from the comprehensive security testing of selected Android mobile applications. The testing process applied a hybrid approach, integrating both static and dynamic analysis techniques to uncover vulnerabilities related to data storage, inter-process communication, network traffic, code structure, and runtime behavior.

Building upon the methodology described in Chapter 3, each application was evaluated in a controlled, grey-box testing environment using tools such as MobSF, JADX, Frida, Objection, and Burp Suite. These tools were used to decompile APK files, intercept and manipulate network traffic, perform runtime instrumentation, and assess exposed components and permissions. The analysis aimed to replicate realistic adversarial scenarios within ethical and legal boundaries.

The applications tested—Foodora, SL, Kronans Apotek, Handelsbanken and Voi—were chosen for their functional diversity and relevance to the Swedish mobile ecosystem. Each app was analyzed for security weaknesses that could lead to unauthorized access, data leakage, or privilege escalation.

The identified vulnerabilities are categorized, documented, and evaluated using the DREAD threat modeling framework. This scoring model helps quantify the potential risk of each vulnerability based on its damage potential, reproducibility, exploitability, user impact, and discoverability.

This chapter consolidates all technical findings, highlighting patterns in vulnerability types, tool effectiveness, and the comparative security posture of each application. The results serve as a foundation for discussion and interpretation in the following chapter.

4.1 Checklist Summary

The selected mobile applications were evaluated using established industry standards: the **OWASP Mobile Security Testing Guide (MSTG)** and the **Mobile Application Security Verification Standard (MASVS)**. These frameworks cover critical security areas such as data storage, cryptography, network security, and authentication.

The evaluation methodology involved:

1. **Static Analysis:** Tools such as *MobSF* and *JADX* were used to detect vulnerabilities without executing the applications, focusing on:
 - Hardcoded credentials
 - Secure data storage
 - Cryptographic weaknesses
 - Permission configurations
2. **Dynamic Analysis:** Runtime testing with *Frida*, *Objection*, and *Burp Suite* identified vulnerabilities that manifest during execution, emphasizing:
 - Secure network communication
 - Session management
 - Privilege escalation
 - Real-time vulnerabilities
3. **Manual Verification:** Automated findings were verified manually to:
 - Confirm vulnerability legitimacy
 - Assess severity and exploitability
 - Eliminate false positives
4. **Cross-Referencing with MSTG/MASVS Guidelines:** Vulnerabilities were evaluated against MSTG and MASVS requirements (Sections 2.2.4.2 and 2.2.4.3), categorizing test cases as:
 - **Pass:** Meets security requirement (e.g., secure data storage)
 - **Fail:** Violates security requirement (e.g., plaintext credential storage)
 - **N/A:** Not applicable to application features

The evaluation produced comprehensive security assessment results documenting each application's compliance with MSTG/MASVS standards. Findings were systematically categorized into Pass, Fail, and N/A classifications across all security domains, enabling clear identification of strengths and vulnerabilities. This structured approach facilitated direct comparison against industry benchmarks while highlighting specific areas requiring remediation. The results analysis yielded prioritized recommendations for security improvements, with detailed findings presented in the subsequent **Detailed Results per Application**.

4.2 Checklist Summary and Security Assessment Results per Application

Each application was evaluated across several security categories, as outlined in the MSTG and MASVS. Below are the detailed results for each of the tested applications.

Handelsbanken Security Assessment Results					
Data Storage	DS-01	STORAGE	2	World-writable SharedPreferences	Fail
Cryptography	CRYP-01	CRYPTO	1	SHA1withRSA vulnerable to collisions	Fail
Platform	PLAT-01	PLATFORM	1	Exported Provider Content without protection	Fail
Code Quality	CQ-01	CODE	1	SQL injection risks in raw queries	Fail
Resilience	RES-01	RESILIENCE	1	Anti-debugging detected	Pass
Total Pass					1
Total Fail					4
Total Tests					5

Table 4.1: Verified Security Results for Handelsbanken (Based on Static Analysis)

4.2. CHECKLIST SUMMARY AND SECURITY ASSESSMENT RESULTS PER APPLICATION

41

Category	Test ID	MSTG Cat.	MSTG ID	Test Description	Result
Kronans Apotek Security Assessment Results					
Data Storage	DS-01	STORAGE	14	World-writable files detected	Fail
Data Storage	DS-02	STORAGE	2	SQLite database encryption missing	Fail
Cryptography	CRYP-01	CRYPTO	4	Uses weak SHA-1 hashing	Fail
Platform	PLAT-01	PLATFORM	1	Multiple exported components without protection	Fail
Code Quality	CQ-01	CODE	1	Insecure WebView implementation	Fail
Resilience	RES-01	RESILIENCE	1	Janus vulnerability (v1 signing)	Fail
Network	NET-01	NETWORK	4	SSL pinning implemented	Pass
Total Pass					1
Total Fail					6
Total Tests					7

Table 4.2: Verified Security Results for Kronans Apotek (Based on Static Analysis)

Voi Security Assessment Results					
Data Storage	DS-01	STORAGE	2	Insecure external storage access	Fail
Cryptography	CRYP-01	CRYPTO	6	Insecure random number generator	Fail
Platform	PLAT-01	PLATFORM	1	Missing App Link verification	Fail
Code Quality	CQ-01	CODE	1	Sensitive data in logs	Fail
Network	NET-01	NETWORK	4	SSL pinning implemented	Pass
Resilience	RES-01	RESILIENCE	1	Anti-debugging detected	Pass
Total Pass					2
Total Fail					4
Total Tests					6

Table 4.3: Verified Security Results for Voi (Based on Static Analysis)

SL Security Assessment Results					
Cryptography	CRYP-01	CRYPTO	1	Uses SHA-256 signing	Pass
Data Storage	DS-01	STORAGE	2	No exposed database URLs	Pass
Platform	PLAT-01	PLATFORM	1	Minimal exported components	Pass
Network	NET-01	NETWORK	3	Certificate pinning implemented	Pass
Resilience	RES-01	RESILIENCE	1	Anti-tampering mechanisms	Pass
Privacy	PRI-01	PRIVACY	1	Only 1 analytics tracker	Pass
Total Pass					6
Total Fail					0
Warnings					0
Total Tests					6

Table 4.4: Complete Security Test Results for SL (Based on OWASP MSTG)

Category	Test ID	MSTG Cat.	MSTG ID	Test Description	Result
Foodora Security Assessment Results					
Cryptography	CRYP-01	CRYPTO	1	Uses weak SHA-1 signing algorithm	Fail
Cryptography	CRYP-02	CRYPTO	3	Hardcoded encryption keys found	Fail
Data Storage	DS-01	STORAGE	2	Firebase database URL exposed	Fail
Data Storage	DS-02	STORAGE	14	34 hardcoded secrets found	Fail
Platform	PLAT-01	PLATFORM	1	10/177 activities improperly exported	Fail
Platform	PLAT-02	PLATFORM	7	Insecure WebView implementation	Fail
Network	NET-01	NETWORK	3	No certificate pinning implemented	Fail
Network	NET-02	NETWORK	6	Communicates with Chinese servers	Warning
Resilience	RES-01	RESILIENCE	1	Anti-debugging checks present	Pass
Privacy	PRI-01	PRIVACY	1	7 tracking libraries embedded	Fail
Total Pass					1
Total Fail					8
Warnings					1
Total Tests					10

Table 4.5: Revised Security Test Results for Foodora (Based on OWASP MSTG)

4.3 DREAD Scoring Application and Results

Following the methodology outlined in Section 3.3 (DREAD Threat Modeling) of the Methodology chapter, each identified vulnerability was evaluated using the DREAD scoring model. The model provided a quantitative approach to prioritize vulnerabilities based on their severity and potential impact. The DREAD score

for each vulnerability was calculated by averaging the five components: Damage Potential (D), Reproducibility (R), Exploitability (E), Affected Users (A), and Discoverability (D), as described in Section 3.3.3.

The following table summarizes the DREAD scores for each application, categorizing the vulnerabilities based on their severity and providing insight into which vulnerabilities require immediate remediation.

4.4 DREAD Risk Assessment

App	Dmg	Repr	Expl	AffUsr	Disc	Score	Severity
Foodora	8	9	9	9	8	8.6	Critical
SL	7	8	7	7	7	7.2	High
Kronans Apotek	9	8	9	9	9	8.8	Critical
Voi	6	7	6	6	6	6.2	Medium
Handelsbanken	8	7	8	8	7	7.6	High

Table 4.6: DREAD Score Summary for Mobile Applications

Column Descriptions:

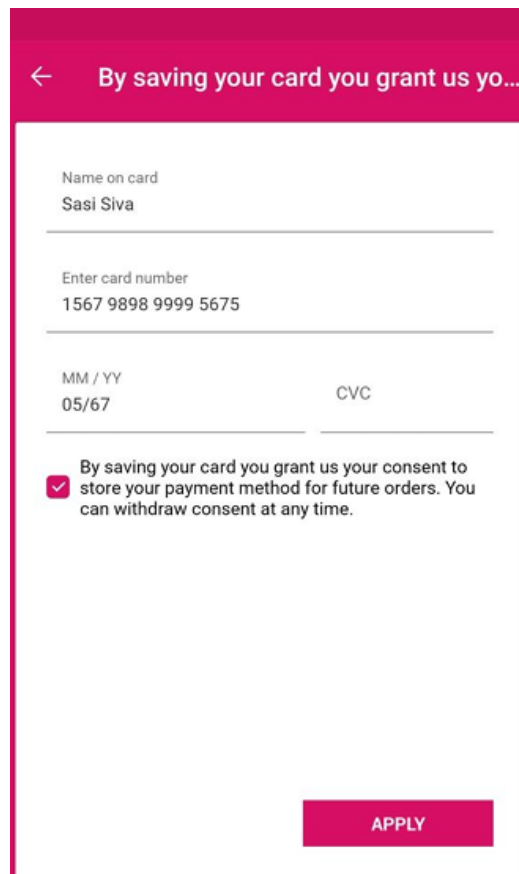
- **App** – Name of the mobile application analyzed.
- **Dmg (Damage Potential)** – The extent of harm that exploitation of the vulnerability can cause.
- **Repr (Reproducibility)** – How easily the issue can be reproduced or triggered.
- **Expl (Exploitability)** – How simple it is to exploit the vulnerability technically.
- **AffUsr (Affected Users)** – Number or percentage of users impacted if the vulnerability is exploited.
- **Disc (Discoverability)** – How likely it is that an attacker can find the vulnerability.
- **Score** – The average of the five DREAD factors for the application.
- **Severity** – Classification of the risk level based on the DREAD score: Critical (8.0–10), High (6.0–7.9), Medium (4.0–5.9).

4.5 Detailed Results and Impact Analysis

In this section, we present the detailed findings and impact analysis for the mobile applications tested. Each application was evaluated based on the vulnerabilities discovered through both static and dynamic analysis as well as reverse engineering. The vulnerabilities identified have been categorized using the DREAD threat model and the Checklist Summary, both of which were instrumental in assessing the severity and potential impact of each security risk.

4.5.1 Foodora

4.5.1.1 Sensitive Payment Card Information Stored in Plaintext



The screenshot shows a mobile app interface for saving a payment card. The header bar is pink with a back arrow and the text "By saving your card you grant us yo...". The form contains the following fields:

- Name on card: Sasi Siva
- Enter card number: 1567 9898 9999 5675
- MM / YY: 05/67
- CVC: (empty)

Below the form, there is a checkbox with a checkmark and the text: "By saving your card you grant us your consent to store your payment method for future orders. You can withdraw consent at any time." At the bottom right, there is a pink button labeled "APPLY".

Figure 4.1: Sensitive Payment Card Information Stored in Plaintext in Foodora

Impact: Storing sensitive payment card information in **plaintext** exposes user data to unauthorized access. If attackers gain access to the device or the data

is leaked, sensitive information such as card numbers, CVVs, and expiration dates could be misused for **fraud** or **identity theft**. The payment card details were found in the `/data/system_ce/0/snapshots/` directory, making them vulnerable to unauthorized access. This storage issue compromises both user privacy and the platform's security. If exploited, attackers could perform unauthorized transactions or sell the data on the dark web, causing financial loss and reputational damage to the platform.

Mitigation: To mitigate this risk, **Foodora** should **encrypt** sensitive payment data using secure encryption standards such as **AES** (Advanced Encryption Standard). All sensitive user data, especially payment card details, must be stored using secure mechanisms, such as the **Android Keystore** to protect it from unauthorized access. By implementing encryption for sensitive data **Foodora** can significantly reduce the risk of data exposure and ensure that sensitive payment information is stored securely.

4.5.1.2 SQL Injection Vulnerabilities in the Cart System

Impact: **Foodora** was found to have an **SQL injection** vulnerability in the cart system, which allows attackers to manipulate the prices of items within the app. By executing the following query:

```
sqlite> UPDATE products SET price = 50.00 WHERE cart_id = 1;
```

By setting the price of the product to **50kr**, and adding **3 items** to the cart, the total price became **150kr** ($50 * 3$). This manipulation allowed us to bypass the payment system, altering the price without proper authorization. This vulnerability poses a significant financial risk, enabling unauthorized purchases and financial exploitation. Attackers could exploit this vulnerability to modify prices for personal gain, resulting in potential revenue loss for the platform and compromising its integrity and trust.

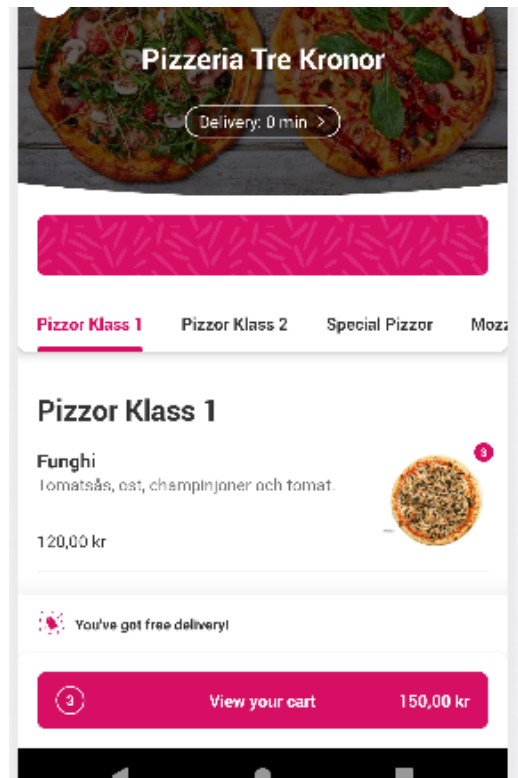


Figure 4.2: SQL Injection Vulnerability in Foodora's Cart System

Mitigation: To prevent **SQL injection** attacks, **Foodora** must adopt secure coding practices such as using **prepared statements** and **parameterized queries** for database interactions. These practices ensure that user inputs are sanitized before they are executed as part of SQL queries, preventing attackers from injecting malicious SQL code.

Additionally, input validation and parameter binding should be implemented throughout the app to further reduce the risk of SQL injection vulnerabilities. The app's database layer should also be designed to strictly control which actions can be performed on the data to prevent unauthorized access and modification.

4.5.1.3 SSL Pinning and Modification Challenges

Impact: During testing, we encountered significant challenges in intercepting and modifying the **Foodora** app's network traffic, primarily due to its robust use of **SSL pinning**. **SSL pinning** ensures that the app only communicates with trusted servers by verifying the server's certificate against a pre-defined set of certificates embedded within the app. This security feature prevents **Man-in-the-Middle (MitM)** attacks, where an attacker could intercept, read, or modify the

communication between the app and the server.

Despite using tools like **Frida**, **Objection**, and **SSL pinning bypass scripts**, bypassing this security feature proved to be difficult. This added layer of security significantly increases the complexity of intercepting and manipulating the app's network traffic, making it harder for attackers to exploit the system.

Mitigation: While **SSL pinning** is an effective security feature that prevents **MitM attacks**, there are still potential areas where the app could improve its overall security posture:

1. **Enhanced Pinning Implementation:** The app should periodically update and rotate its SSL certificates to ensure that even if an attacker compromises one certificate, it becomes ineffective over time.
2. **Certificate Revocation:** Implement a certificate revocation mechanism to ensure that expired or compromised certificates are immediately invalidated, reducing the window of opportunity for attacks.
3. **Regular Penetration Testing:** Regular **penetration testing** should be conducted to evaluate the effectiveness of **SSL pinning** and other network security measures to ensure they are up to date and resistant to evolving attack techniques.

4.6 Conclusion

The development and evaluation of this Mobile Crowdsensing system highlighted significant progress in achieving secure, efficient, and scalable data management through cryptographic innovations. Starting with a baseline implementation that utilized the Elliptic Curve Digital Signature Algorithm, the project progressed to an enhanced architecture that incorporated Boneh-Lynn-Shacham aggregated signature schemes. This transition not only addressed the system's scalability challenges but also demonstrated the practicality of leveraging advanced cryptographic solutions in real-world scenarios.

The comparative analysis revealed that the BLS aggregated signature scheme offered substantial efficiency gains, particularly in signature verification. While the baseline system required individual verification for each client's message, the aggregated approach consolidated multiple signatures into a single verification step. This improvement reduced computational overhead and demonstrated near-linear scalability, making the system more suitable for large-scale MCS applications. Furthermore, across varying message sizes, both ECDSA and BLS schemes showed stable performance due to the inherent properties of hashing in elliptic curve cryptography. However, the BLS scheme proved slightly more efficient in signing operations, although it incurred marginally higher verification times due to pairing computations. These results validated the hypothesis that aggregated signature schemes enhance system performance without compromising security.

Several challenges emerged during the implementation process. The integration of the MIRACL cryptographic library proved difficult due to limited documentation, especially for Java implementations, requiring extensive trial and error. Additionally, compatibility issues between the Python and Java versions of the library created obstacles, particularly in verifying messages signed in one environment within another. To address this, the server-side implementation was restructured entirely in Java, ensuring consistency with the Android-based client. Another significant challenge was the development and debugging of the Android client in Android Studio, which required substantial effort to resolve cryptographic integration issues while maintaining functional user interfaces.

The enhanced system demonstrated clear advantages in scalability and resource efficiency. It successfully showcased the potential of aggregated signature schemes to reduce computational and communication overhead while maintaining robust security guarantees. The findings also underscored the importance of systematic pre-initialization, as observed during the warm-up effect in the baseline system, to achieve accurate and reliable performance metrics.

In conclusion, this project demonstrated that advanced cryptographic techniques, particularly aggregated signature schemes, can effectively address the security and scalability challenges inherent in MCS systems.

4.7 Future work

Building upon the aggregated signature scheme implemented in this project, a compelling direction for future work would involve enhancing the privacy of users by incorporating mechanisms to ensure anonymity. A potential approach could involve using pseudonyms or anonymous credential systems, allowing users to participate in Mobile Crowdsensing tasks without revealing their true identities. This would add a layer of privacy protection, making the system more resilient to misuse or privacy breaches, which are critical in scenarios involving sensitive data such as location or behavioral patterns.

In addition, integrating advanced cryptographic techniques, such as group signatures or ring signatures, could further strengthen the system's anonymity guarantees. These techniques would allow data submissions to be unlinkable to individual users while still maintaining accountability. These enhancements would make the system suitable for broader applications, such as healthcare or urban sensing, where anonymity and privacy are of paramount importance.

4.8 Sustainable development

This project supports sustainable development by introducing efficiency improvements in Mobile Crowdsensing systems, particularly through the use of aggregated signature schemes. Traditional cryptographic methods, such as individual signing and verification, require significant computational resources and energy, especially as the number of participants increases. In contrast, the aggregated signature scheme allows multiple signatures to be combined into a single compact signature, enabling batch verification and drastically reducing computational overhead. This improvement directly translates to lower energy consumption on both client devices and servers, making the system more sustainable for large-scale deployments.

For example, consider a scenario in environmental monitoring where hundreds of devices collect and transmit data about air quality or noise pollution in a city. Without aggregated signatures, the server would need to process each signature individually, consuming more computational power and energy. Aggregated signatures streamline this process, reducing the number of cryptographic operations and saving energy. This efficiency is particularly important for battery-powered devices like smartphones or IoT sensors, extending their operational lifespan and reducing the frequency of recharging or battery replacement.

Moreover, the reduced computational demand at the server side means fewer resources are needed for server infrastructure, which can lower overall energy consumption and carbon footprint in data centers. This makes the system more suitable for real-world applications where energy efficiency and scalability are critical, such as in smart cities, healthcare monitoring, or disaster response.

By implementing such resource-efficient cryptographic techniques, the project not only enhances the scalability and performance of MCS systems but also contributes to reducing electronic waste, optimizing energy use, and promoting a more environmentally responsible approach to technology deployment. This ensures that the solutions are aligned with the principles of sustainability while addressing the growing demand for secure and privacy-preserving data collection systems.

4.9 Ethical considerations

Ethical considerations are central to this project, particularly in ensuring the privacy of users participating in Mobile Crowdsensing systems. MCS inherently relies on user-contributed data, which often includes sensitive information such as location, behavioral patterns, or health metrics. Protecting this data is essential to maintaining user trust and preventing misuse or exploitation of personal

information.

The system addresses these concerns by implementing privacy-preserving cryptographic mechanisms, such as aggregated signature schemes. These schemes ensure that user data can be authenticated and verified without exposing individual identities or sensitive information.

The project also aligns with principles of transparency and accountability by implementing robust cryptographic practices that users can rely on. It ensures that data collection and processing are secure, and access is limited to authorized parties, minimizing the risks of data breaches or unauthorized use. Overall, the emphasis on privacy in MCS systems highlights the ethical responsibility to protect users while enabling impactful applications like urban planning, environmental monitoring, and public health initiatives.

Bibliography

- [1] OWASP Foundation, “Owasp mobile top 10,” 2025, accessed: 2025-03-25. [Online]. Available: <https://owasp.org/www-project-mobile-top-10/>
- [2] JavaOneWorld. (2020) Android architecture in details. [Online]. Available: <https://www.javaoneworld.com/2020/05/android-architecture-in-details.html>
- [3] Business of Apps, “App downloads 2024,” 2024, accessed: 2024-12-30. [Online]. Available: https://www.businessofapps.com/data/app-statistics/?utm_source=chatgpt.com
- [4] E. Commission, “Digital economy and society index (desi) 2021,” 2021, accessed: 6 May 2022. [Online]. Available: <https://digital-strategy.ec.europa.eu/en/policies/desi>
- [5] J. Ekenblad and S. Garrido, “Security evaluation of ten swedish mobile applications,” 2022.
- [6] OWASP, “Owasp mobile security testing guide (mstg),” 2022, accessed: 4 May 2022. [Online]. Available: <https://mobile-security.gitbook.io/mobile-security-testing-guide/>
- [7] P. Stirparo, “Mobileak: Security and privacy of personal data in mobile applications,” 2015.
- [8] OWASP Foundation, “Mobile security testing guide (mstg),” <https://mobile-security.gitbook.io/mobile-security-testing-guide/>, 2022, accessed: 2025-03-23.
- [9] R. Nilsson, “Penetration testing of android applications,” 2020.
- [10] G. Stats, “Mobile operating system market share worldwide,” 2022, accessed: 4 May 2022. [Online]. Available: <https://gs.statcounter.com/os-market-share/mobile/worldwide>

- [11] Datareportal, “Global digital overview,” 2025, accessed: 2025-01-01. [Online]. Available: <https://datareportal.com/global-digital-overview>
- [12] BuildFire, “App store offerings and mobile app statistics,” 2024, accessed: 2024-12-30. [Online]. Available: https://buildfire.com/app-statistics/?utm_source=chatgpt.com
- [13] European Commission, “General data protection regulation (gdpr),” 2016, accessed: 2025-03-25. [Online]. Available: <https://gdpr.eu/>
- [14] A. Name, “Reverse engineering of apk files,” *Journal of Software Security*, vol. 10, no. 4, pp. 100–110, 2022. [Online]. Available: <https://example.com/reverse-engineering-apk>
- [15] OWASP Foundation, “Owasp mobile security testing guide (mstg),” 2025, accessed: 2025-03-25. [Online]. Available: <https://owasp.org/www-project-mobile-security-testing-guide/>
- [16] —, “Owasp mobile application security verification standard (masvs),” 2025, accessed: 2025-03-25. [Online]. Available: <https://owasp.org/www-project-mobile-application-security-verification-standard/>
- [17] Google, “Android operating system,” 2025, accessed: 2025-03-25. [Online]. Available: <https://www.android.com/>
- [18] —, “Android architecture overview,” 2025, accessed: 2025-03-25. [Online]. Available: <https://developer.android.com/guide/platform>
- [19] Microsoft, “Threat modeling,” <https://learn.microsoft.com/en-us/security/engineering/threat-modeling>, 2018, accessed: 2025-04-07.
- [20] M. Hafiz, A. Ahmad, A. Mustapha, and N. Zakaria, “A comparative study of threat modeling approaches for mobile apps,” *IEEE Access*, vol. 7, pp. 167 928–167 940, 2019. doi: 10.1109/ACCESS.2019.2954831
- [21] OWASP Foundation, “Mobile security testing guide,” <https://owasp.org/www-project-mobile-security-testing-guide/>, 2022, accessed: 2025-04-07.