MoSec

A Tool for Evaluating Branchless Banking Applications

SE-801 SPL-3 Final Report

Supervised by
Adwait Nadkarni
Associate Professor
Computer Science Department,
Williams & Mary
Williamsburg, Virginia

Mentored by
Prianka Mandal
PhD. Candidate
Computer Science Department,
Williams & Mary
Williamsburg, Virginia

Internally Supervised by
Dr. Naushin Nower
Professor
Institute of Information Technology,
University of Dhaka

Submitted by
Tashfia Jannath
BSSE-1223
Institute of Information Technology,
University of Dhaka

Submission Date: January 30, 2025

Letter of Transmittal

January 30, 2025

BSSE 4th Year Exam Committee

Institute of Information Technology,

University of Dhaka

Subject: Technical Report Submission - **MoSec:** A Tool for Evaluating Branchless Banking Applications

Sir,

I am submitting the technical report for our Software Project Lab 3, "**MoSec:** A Tool for Evaluating Branchless Banking Applications”.

This report comprehensively covers the project's technical details, including design, implementation, and testing phases. While I’ve tried to do my best, I welcome your feedback for improvement.

Thank you for your consideration.

Sincerely,

Tashfia Jannath

BSSE-1223

Institute of Information Technology

University of Dhaka

—------------------------------

Supervisor’s Signature

Acknowledgement

I would like to express my sincere gratitude to my project supervisor, Dr. Adwait Nadkarni Sir, Assistant Professor, William & Mary, Williamsburg VA, for allowing me to go on with this project and providing unconditional support throughout the project selection phase. I would also like to thank my internal supervisor, Dr. Naushin Nower Ma'am, Professor, Institute of Information Technology, University of Dhaka. Her invaluable guidance and motivation helped me go further in this project. It is an honor to work under her supervision. Finally, I would like to pay my gratitude to my mentor, Prianka Mandal, PhD candidate, William & Mary for being available whenever I needed guidance.

Abstract

Mobile money applications (MMAs) have transformed financial inclusion, providing access to banking and transaction services for millions in developing nations. However, these applications are often built without adhering to robust security standards, leaving users vulnerable to cyber threats such as data breaches, financial fraud, and privacy exploitation. The **MoSec** (Mobile Banking Application Security) project aims to address these concerns by developing a comprehensive analysis framework for evaluating the security of MMAs. This framework integrates both automated static analysis tools and manual auditing processes, enabling the detection of vulnerabilities such as cryptographic misuse, data leakage, and improper configurations. The ultimate goal is to enhance the security landscape for mobile money users by identifying and mitigating critical risks.

Our methodology encompasses two core components: an automated pipeline for performing static analysis and a manual analysis module. The static analysis leverages state-of-the-art tools to identify common vulnerabilities like over-permissions and cryptographic flaws, while the manual analysis focuses on domain-specific issues such as improper SSL configurations and sensitive data exposure. Additionally, the MoSec framework consolidates the findings into an interactive dashboard, allowing developers, regulators, and stakeholders to access detailed reports and implement necessary fixes. By providing a centralized and streamlined tool, MoSec aims to simplify the complex process of MMA security evaluation.

MoSec stands out by adopting a holistic approach to mobile money security, catering specifically to applications widely used in the developing world. The system is designed to adapt to the evolving threat landscape, ensuring ongoing relevance and effectiveness. The project ultimately aspires to build trust in mobile financial systems by equipping developers with actionable insights and empowering users with safer applications. Through its integration of automation, manual oversight, and interactive reporting, MoSec not only identifies vulnerabilities but also lays the groundwork for proactive security enhancements in mobile money ecosystems.

Table of Contents

1. Introduction

Mobile money applications, also known as branchless banking, have revolutionized financial services in developing countries but often suffer from serious security vulnerabilities. Many apps lack proper SSL/TLS encryption, exposing users to threats like MITM attacks, data leaks, and insecure cryptography. MoSec is a tool that combines static and manual analysis to detect these vulnerabilities. The tool will analyze APK files, automate the methodology, and present results via a web dashboard with report generation features.

1.1 Motivation

Mobile money applications have transformed financial inclusion in developing countries, providing vital services to populations without access to traditional banking. However, this innovation comes with significant security risks. Vulnerabilities such as improper SSL/TLS encryption, weak authentication, and data leakage put users' financial data at risk and undermine trust in these systems. Addressing these security gaps is essential to safeguard users and ensure the continued growth of branchless banking. This project aims to develop a tool that automates the detection of these vulnerabilities, providing accurate and actionable insights to enhance mobile money app security.

1. **Improving Financial Security**: Mobile money systems often serve populations most vulnerable to fraud and cybercrime. Ensuring their security is vital for protecting user data and transactions.
2. **Bridging Gaps in Automated Analysis**: Existing automated tools fail to detect critical vulnerabilities or produce false positives, leaving significant risks unaddressed.
3. **Scaling Security Solutions**: Manual analysis is accurate but time-consuming. By combining static and manual analysis in an automated pipeline, this tool offers scalable and efficient security assessments.

4. **Enhancing Trust in Branchless Banking**: Security issues erode user confidence in mobile financial services. Strengthening app security can promote wider adoption and trust in these essential systems.
5. **Compliance and Standardization**: Adopting a framework that classifies vulnerabilities using CWE codes ensures standardized reporting and helps developers align with best practices.
6. **Empowering Developers and Organizations**: By providing detailed reports and actionable insights, the tool helps developers address vulnerabilities proactively, improving the overall quality of mobile money apps.

This project aligns with the global push for secure financial inclusion, addressing critical gaps in mobile app security to protect users and ensure the sustainability of branchless banking systems.

1.2 Scope

The scope of the project is to -

1. **Comprehensive Security Analysis**: Automates the detection of critical vulnerabilities in mobile money applications, such as SSL/TLS misconfigurations, improper cryptographic implementations, and data leakage.
2. **Integrated Methodologies**: Combines static analysis and manual inspection to ensure thorough detection of security flaws, covering the entire app lifecycle (e.g., registration, login, and transactions).
3. **User-Friendly Interface**: Provides a web-based platform for uploading APK files, viewing analysis results, and generating detailed reports.
4. **Vulnerability Classification**: Uses Common Weakness Enumeration (CWE) codes for standardized categorization of detected vulnerabilities.
5. **Scalable and Efficient**: Designed to handle large volumes of APK files with high accuracy, reducing reliance on time-intensive manual analysis.

6. **Focus on Mobile Money Apps**: Targets branchless banking applications, particularly those operating in developing countries where traditional banking infrastructure is limited.
7. **Dynamic and Static Analysis**: Includes tools for analyzing both static code issues and runtime behaviors to ensure comprehensive coverage.
8. **Customizable Reporting**: Generates detailed, exportable reports that can assist developers in prioritizing and addressing security flaws.

1.3 Limitations of the Project

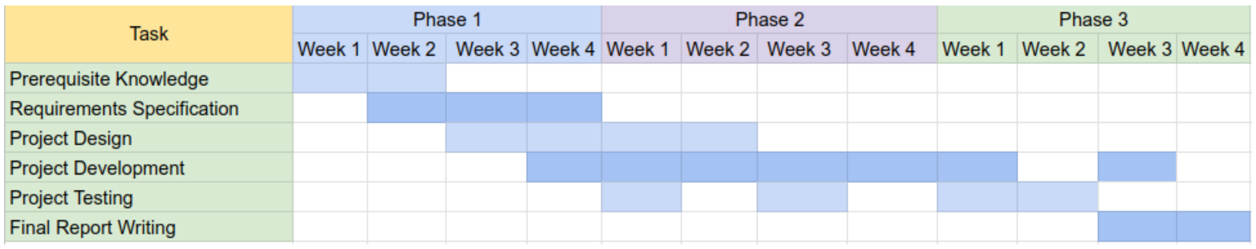
The limitations of the project is to -

1. **Focus on Android**: Limited to Android applications due to the platform's dominance in developing markets; does not analyze iOS apps.
2. **Dependency on APK Accessibility**: Requires access to APK files for analysis, which might not always be available for certain apps or proprietary systems.
3. **Server-Side Security Exclusion**: Focuses primarily on client-side vulnerabilities and cannot comprehensively analyze server-side configurations beyond SSL/TLS endpoints.
4. **Limited Real-World Testing**: Testing findings in real-world conditions might be restricted due to ethical concerns, resource constraints, or access limitations.
5. **Reliance on Known Vulnerability Patterns**: Detects vulnerabilities based on existing patterns and CWE codes but may not identify novel or emerging threats.
6. **Resource Intensity for Manual Analysis**: The manual validation phase, while reducing false positives, may still require significant time and expertise.
7. **Complexity of Advanced Encryption Flaws**: May face challenges detecting highly sophisticated cryptographic attacks or vulnerabilities not evident from code alone.
8. **No Direct Remediation**: The tool identifies vulnerabilities but does not provide automatic fixes or apply security patches.

This scope outlines the project's ambitious goals for improving mobile money application security, while the limitations provide a realistic perspective on the challenges and boundaries of the solution.

1.4 Project Timeline

The estimated timeline of MoSec is given below:



Task	Phase 1				Phase 2				Phase 3			
	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4
Prerequisite Knowledge	▨	▨										
Requirements Specification		🟦	🟦	🟦								
Project Design			▨	▨	▨	▨						
Project Development				🟦	🟦	🟦	🟦	🟦	🟦		🟦	
Project Testing					▨		▨		▨	▨		
Final Report Writing											🟦	🟦

Figure-1: Project Timeline (Estimated)

1.5 Purpose of Document

The purposes of this document are-

- Identify and analyze the requirements
- Design the architecture
- Design the test plan
- Describe methodology
- Reduce the development effort
- Improve understanding

2. Project Description

This project focuses on enhancing the security of mobile money applications, which play a vital role in financial inclusion in developing countries. By combining automated static analysis with manual inspection, it aims to identify critical vulnerabilities such as SSL/TLS misconfigurations, improper certificate validation, insecure cryptographic implementations, and data leakage. The project involves creating a comprehensive tool that accepts APK files, disassembles and analyzes their code, tests server-side SSL/TLS configurations, and examines data flows for potential weaknesses. Results are presented through a user-friendly web interface with detailed, CWE-classified reports, enabling developers to address issues efficiently. This solution bridges the gap between accuracy and scalability, offering actionable insights to improve the security posture of branchless banking systems while fostering trust in mobile financial services.

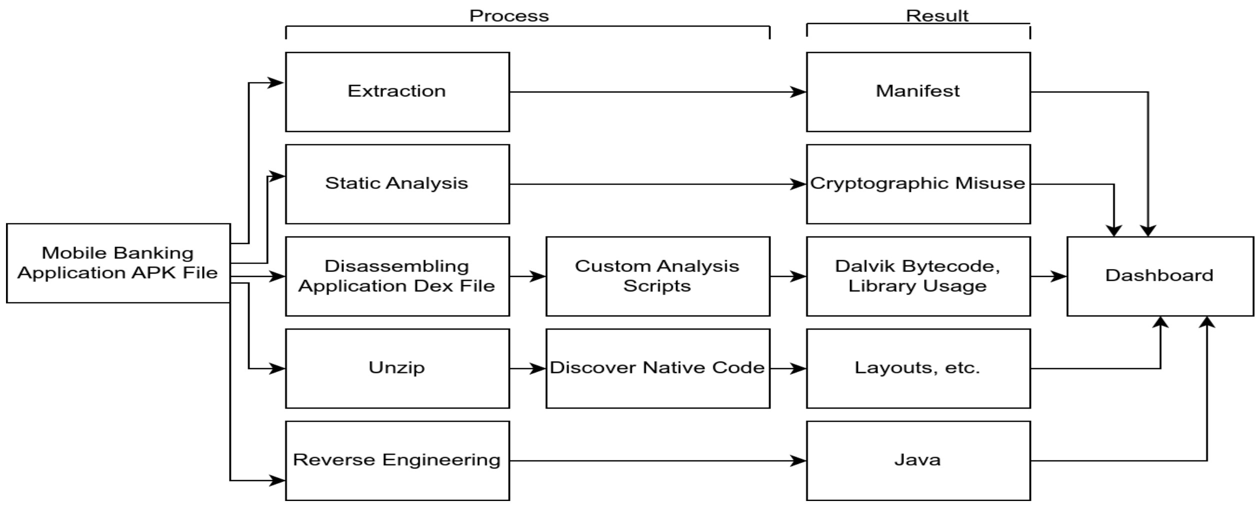


Figure-2: Technical Overview of MoSec

Figure-2 is the technical overview of MoSec.

2.1. Quality Function Deployment

Quality Function Deployment (QFD) is a technique that translates the needs of the customer into technical requirements for the software. With respect to this project, the following requirements are identified -

2.1.1 Normal Requirements

1. Users will be able to upload the apk files as it is an essential starting point for the analysis process.
2. Users expect the tool to provide comprehensive metrics and insights to understand the applications better.
3. Users can do apk file extraction, custom static analysis, ssl/tls server testing, reverse engineering, data leakage detection, manual analysis automation and CWE analysis.

2.1.2 Expected Requirements

1. Allowing for the addition of new analysis methods or libraries without major redesigns.
2. Reports can be downloaded as PDF.

2.1.3 Exciting Requirements

1. Comparing analyzed apps' security levels against industry benchmarks or peer applications.
2. Providing APIs or plugins to integrate with development tools, enabling automatic security checks during app builds.

2.2 Usage Scenario

1. Uploading APK Files for Comprehensive Security Analysis

Developers or security teams can upload APK files through a web interface for analysis. The system processes these files by extracting their manifest, permissions, and metadata using apktool. This initial step provides a high-level overview of the app's structure and identifies any potential red flags before further analysis. It's an easy starting point for users with no technical expertise in manual APK disassembly.

2. Detecting Vulnerabilities through Static Analysis

The custom static analysis tool leverages the Androguard API to examine the app's code for vulnerabilities. It checks for SSL/TLS misconfigurations, improper cryptographic implementations, and access control issues. For example, if the app uses insecure encryption algorithms like MD5 or fails to validate SSL certificates properly, these flaws are flagged. This step is essential for identifying weaknesses embedded in the codebase.

3. Assessing Server-Side SSL/TLS Configurations

For apps that communicate with remote servers, the system performs SSL/TLS server testing using tools like the Qualys SSL Server Test. This ensures that the server-side configuration is secure, with strong cipher suites, valid certificates, and protection against common attacks like man-in-the-middle (MITM). This functionality helps developers validate server security in addition to the app's code.

4. Automating Manual Analysis for Critical Libraries

To save time and improve accuracy, the system automates manual analysis tasks. Using Baksmali, the tool disassembles the Dalvik bytecode to detect cryptographic libraries and dangerous third-party libraries. Regular expressions and pattern matching scripts identify insecure coding practices or untrusted modules, simplifying what would otherwise require manual inspection by a security analyst.

5. Lifecycle Tracing of Sensitive Data Handling

The tool performs a control flow analysis to trace the app's lifecycle, starting from the onCreate() method to sensitive functionalities like user registration, login, and money transfer. By analyzing how sensitive data such as authentication tokens or financial records is handled during these operations, the tool ensures that the app is not exposing private information to attackers.

6. Reverse Engineering for In-Depth Vulnerability Inspection

Security analysts can use the reverse engineering feature to gain deep insights into the app's functionality. By leveraging the JEB Decompiler, the tool examines how cryptographic routines, session management, and data flows are implemented. This step is particularly valuable for auditing closed-source apps or verifying compliance with security standards.

7. Identifying Sensitive Data Leakage

To safeguard user privacy, the tool checks for data leaks during app execution. It analyzes whether sensitive information, such as personal data or financial records, is being logged insecurely or transmitted via unencrypted channels like plaintext HTTP. This detection step is critical for ensuring compliance with data protection regulations.

8. Compiling and Visualizing Results on a Dashboard

Once the analysis is complete, the tool presents results on a web-based dashboard. Vulnerabilities are categorized using CWE (Common Weakness Enumeration) codes, making it easier for developers to understand the severity and type of each issue. Users can also download a detailed report summarizing the findings and recommendations for remediation.

3. Scenario Based Modeling

Scenario-based modeling is an approach used in software development and design to create models that depict how a system behaves or reacts in various real-world situations or scenarios. These scenarios are typically based on user interactions, events, or conditions and help developers understand and visualize how the software will function in different contexts. By simulating these scenarios, developers can better identify potential issues, make informed design decisions, and ensure that the software meets user requirements across different usage contexts.

3.1 Use Case Diagram

A use case diagram is a visual representation in software engineering that illustrates how a system interacts with its external entities or actors to achieve specific goals or tasks. It shows the relationships between use cases (representing system functionalities) and actors (representing external users or systems). Use case diagrams are typically created through requirements gathering, where system functionalities and user interactions are identified, and then diagrammed using specialized software modeling tools.

Level 0 of Use Case Diagram

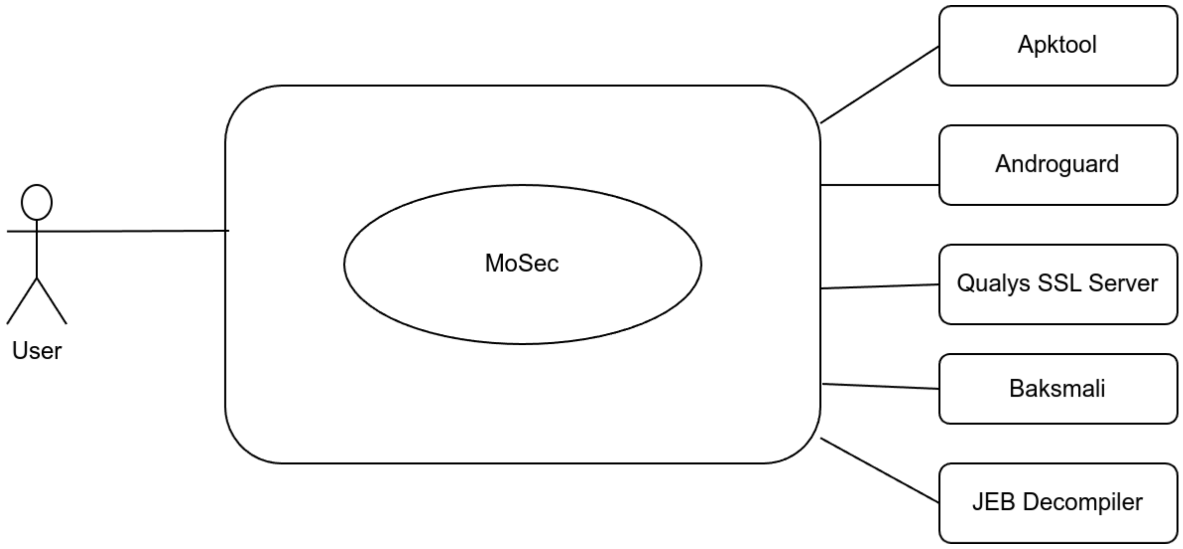


Figure 3: Use Case Diagram L-0

Name: MoSec

Primary actors: User

Secondary actors: Apktool, Androguard, Qualys SSL Server, Baksmali, JEB Decompiler

Goal in Context: The diagram shown above represents the whole tool for detecting vulnerabilities of mobile banking applications.

Level 1 of Use Case Diagram

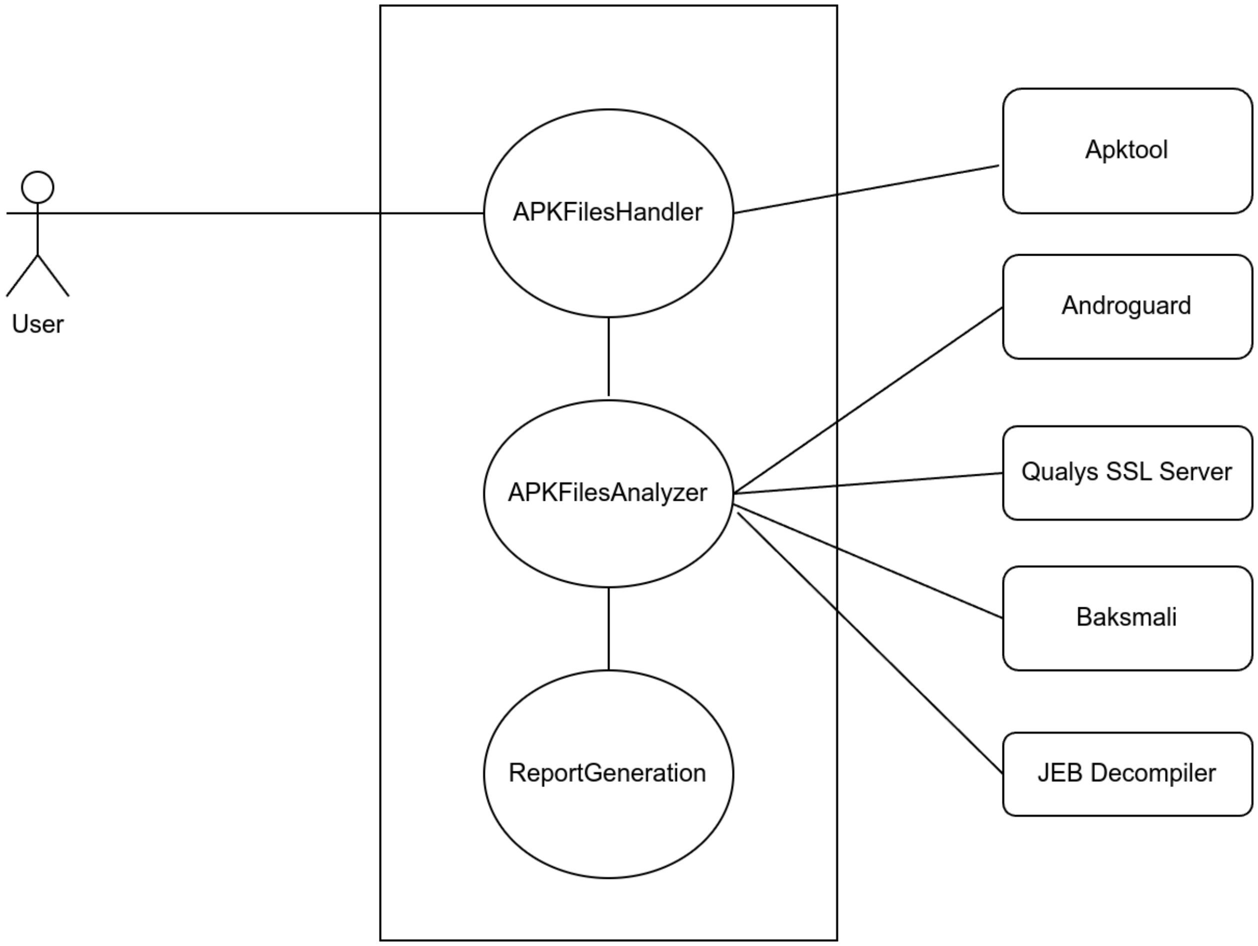


Figure-4: Use Case Diagram L-1

Name: MoSec

Primary actors: User

Secondary actors: Apktool, Androguard, Qualys SSL Server, Baksmali, JEB Decompiler

Goal in Context: The diagram shows the modules of MoSec. MoSec consists of 3 modules. The modules can be elaborated as follows:

1. **APKFilesHandler**: This module serves as the entry point for MoSec's functionality.The user will upload APK files via the web interface. The backend will use apktool to extract the application manifest, permissions and metadata to provide a high-level overview of the app's structure and functionality.
2. **APKFilesAnalyzer**: This module handles detection of vulnerabilities in apk files.

3. **ReportGeneration**: The results of both the static and manual analysis will be displayed on a dashboard. This includes identified vulnerabilities such as SSL/TLS protocol errors, cryptographic weaknesses, data leakage, and authentication flaws.

Level 1.1.1 of Use Case Diagram

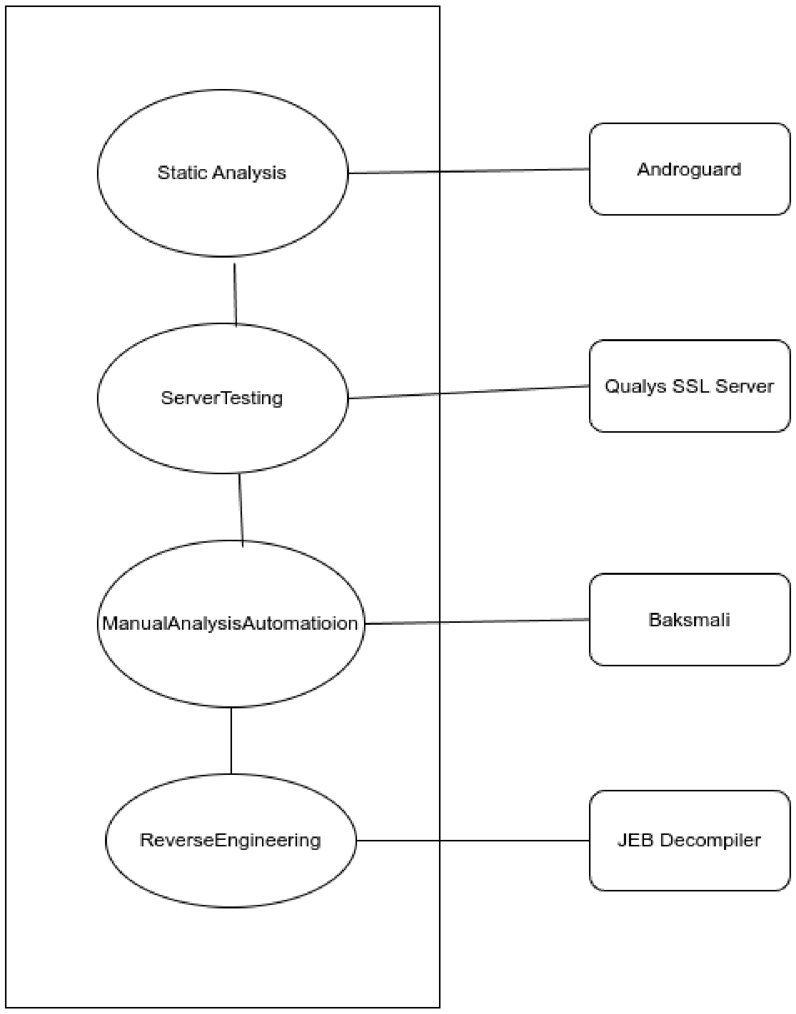


Figure-5: Use Case Diagram L-1.1.1

Name: MoSec

Primary actors: User

Secondary actors: Apktool, Androguard, Qualys SSL Server, Baksmali, JEB Decompiler

Goal in Context: The diagram shows the modules of APKFilesAnalyzer. APKFilesAnalyzer consists of 4 modules. The modules can be elaborated as follows:

1. **StaticAnalysis:** This module analyzes the SSL/TLS certificate validation, improper cryptography, and access control issues.
2. **ServerTesting**: It will examine the server-side configuration for protocol strength, cipher choices, and certificate validity.
3. **ManualAnalysisAutomation**: It automates the detection of cryptography libraries and dangerous third-party libraries using regex and pattern matching scripts.
4. **ReverseEngineering**: It verifies the presence of vulnerabilities in sensitive functionalities, including authentication, cryptographic procedures, and data handling.

5. Class-based Modeling

This chapter describes the class-based modeling of MASS. Class-based modeling represents the objects that the system will manipulate, the operations (also called methods or services) that will be applied to the objects to affect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. The elements of a class-based model include classes and objects, attributes, operations, class-responsibility-collaborator (CRC) models, collaboration diagrams, and packages.

5.1 Analysis Classes

After identifying the nouns from the scenario, I filtered nouns belonging to the solution domain using General Classification (External entities, Things, Events, Roles, Organizational units, Places, and Structures). Nouns selected as a potential class were filtered using Selection Criteria (Retained information, Needed services, Multiple attributes, Common attributes, Common operations, and Essential requirements). After performing analysis on potential classes, I have found the following analysis classes -

1. APKFile
2. StaticAnalyzer
3. AutomatedAnalyzer
4. ReportGenerator
5. Dashboard
6. SSLAnalyzer
7. CryptographyAnalyzer
8. DataLeakageDetector

5.2 Class Cards

APKFile	
Attributes	**Methods**

- file_name
- file_path
	+ validate_file_input()
	+ extract_metadata()

- metadata - permissions	+ list_permissions()
Responsibilities	**Collaborators**
● Validate the APK file input provided by the user. ● Extract metadata such as package information, version, etc. ● List permissions declared in the APK.	● StaticAnalyzer ● AutomatedAnalyzer

Table 2: CRC Card of APKFile

StaticAnalyzer	
Attributes	**Methods**

- apk_file
- analysis_results
	+ perform_static_analysis()
	+ detect_SSL-issues()
	+ detect_crypto_misuse()

Responsibilities	**Collaborators**
● Perform static analysis of the APK file. ● Detect misconfigurations or issues related to SSL. ● Identify improper usage of cryptographic APIs.	● APKFile ● SSLAnalyzer ● CryptographyAnalyzer

Table 3: CRC Card of StaticAnalyzer

AutomatedAnalyzer	
Attributes	**Methods**

- apk_file
- analysis_results
	+ perform_automated_analysis()
	+ trace_sensitive_libraries()
	+ analyze_lifecycle()

Responsibilities	**Collaborators**

• Perform automated analysis of the APK for complex security issues. • Identify sensitive libraries used in the app. • Analyze the application lifecycle for potential leaks.	• APKFile • DataLeakageDetector

Table 4: CRC Card of AutomatedAnalyzer

ReportGenerator	
Attributes	**Methods**

- analysis_data
- automated_results
- report
	+ compile_results()
	+ categorize_by_cwe()
	+ generate_reports()

Responsibilities	**Collaborators**
• Compile results from static and automated analysis. • Categorize vulnerabilities based on CWE standards. • Generate a detailed analysis report.	• Dashboard

Table 5: CRC Card of ReportGenerator

Dashboard	
Attributes	**Methods**

- uploaded_files
- analysis_data
- user_session
	+ display_results()
	+ allow_file_upload()

Responsibilities	**Collaborators**

● Serve as the main interface for the user to interact with the system. ● Display results of the analysis. ● Handle file uploads from users.	● APKFile ● ReportGenerator

Table 6: CRC Card of Dashboard

SSLAnalyzer	
Attributes	**Methods**
- certificate_details - protocol_strength	+ analyze_SSL_Configs()
Responsibilities	**Collaborators**
● Analyze SSL configurations for vulnerabilities. ● Evaluate the strength of protocols used in the APK.	● StaticAnalyzer

Table 7: CRC Card of SSLAnalyzer

CryptographyAnalyzer	
Attributes	**Methods**
- crypto_apis_used - misconfigurations	+ analyze_crypto_apis()
Responsibilities	**Collaborators**

• Analyze cryptographic APIs used in the app. • Identify potential misconfigurations or misuse.	• StaticAnalyzer

Table 8: CRC Card of CryptographyAnalyzer

DataLeakageDetector	
Attributes	**Methods**
- data_flows - detected_leaks	+ detect_leaks()
Responsibilities	**Collaborators**
• Detect data leaks and sensitive information exposures. • Analyze data flows within the application.	• AutomatedAnalyzer

Table 9: CRC Card of DataLeakageDetector

5.3 Class-Responsibility-Collaborator Diagram

The CRC Diagram for MoSec is given below

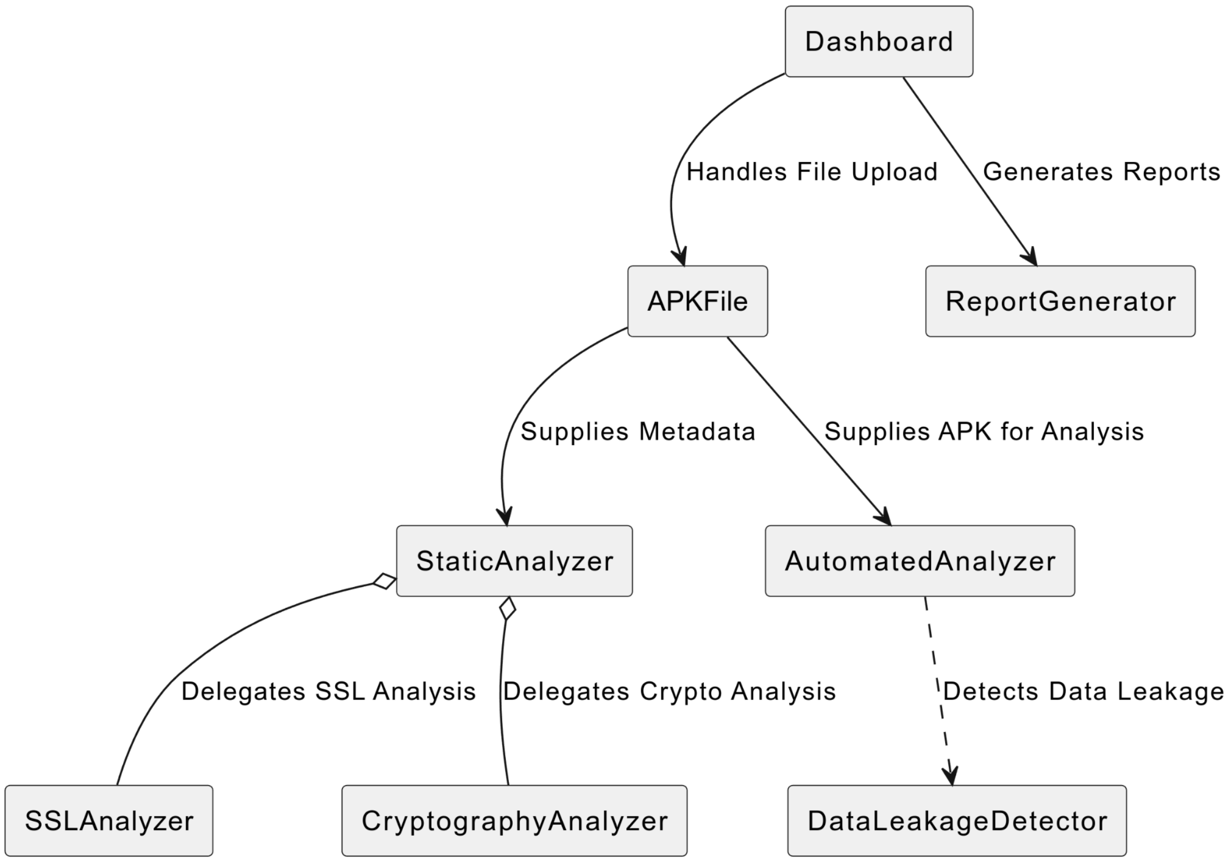


Figure-7: CRC Diagram of MoSec

6. Software Design Architecture

Architectural design is a visual representation in software engineering that outlines the high-level structure and organization of a software system. It focuses on defining the major components or modules of the system, their interactions, and the overall system's architecture.

6.1 Architectural Context Design

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. Systems that interoperate with the target system are represented as -

- **Superordinate systems—** Those systems that use the target system as part of some higher-level processing scheme.
- **Subordinate systems—** Those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- **Peer-level systems—** Those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system).
- **Actors—** Entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

The architectural context diagram for MoSec is given below.

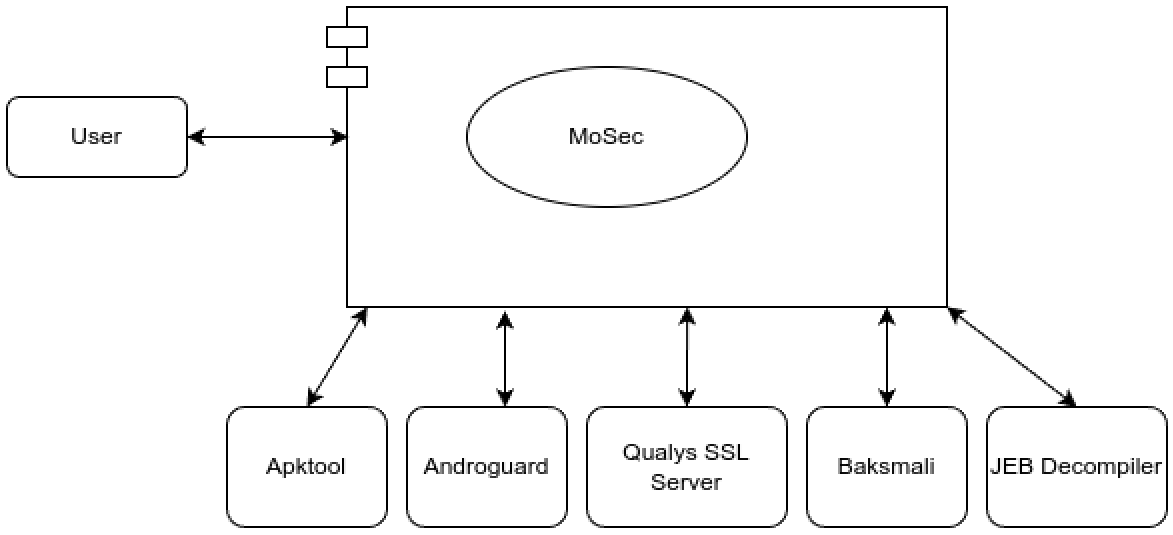


Figure-8: Architectural Context Diagram of MoSec

Representation of MoSec in Architectural Context Diagram:

- **Subordinate Systems—** The apps are analyzed via different tools.
- **Actor—*Users.*** The primary actors use MoSec to provide configurations, view reports and analyze results.

6.2 Top Level Components

Overall architectural structure with top level components is illustrated below. Note that, Top Level Components diagrams show a very high level view of our system. It is further refined into architecture in the following sections.

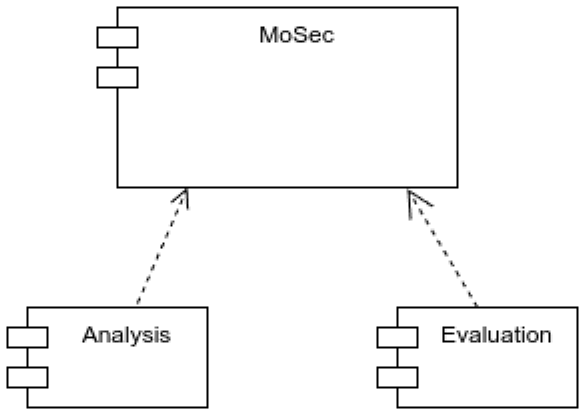


Figure-9: Top Level Components of MoSec

6.3 Instantiation of the Architecture

The top level components are refined into smaller components. We also apply architecture on the components to make a maintainable collection of components. The architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Component: Analysis

For the Analysis component, we opted to use the Pipe and Filter architecture. The Pipe and Filter architecture is a software design pattern that organizes a system as a series of processing elements (filters) connected by communication channels (pipes). It is particularly useful for designing applications that involve processing streams of data in a linear and sequential manner. Each filter performs a specific task on the incoming data and passes the results to the next filter through the pipes, creating a data flow pipeline.

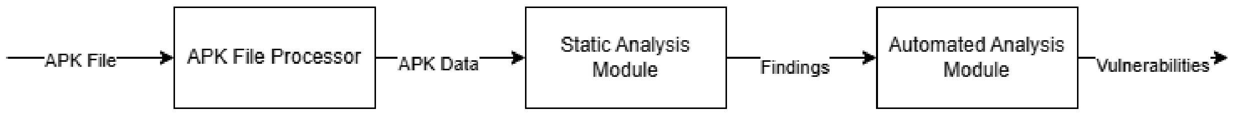


Figure-10: Architecture Instantiation on Component 'Analysis'

Component: Evaluation

For the Evaluation component also, we opted to use the Pipe and Filter architecture.

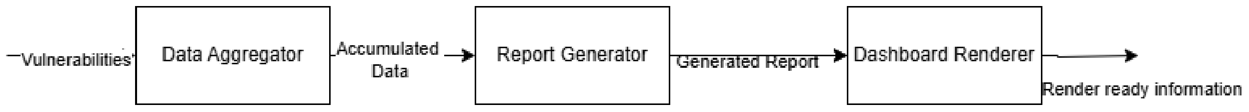


Figure-11: Architecture Instantiation on Component 'Evaluation'

6.4 Mapping Requirements to Software Architecture

The mapping of the derived components with the requirements can be shown as below.

Requirements	Components
1. Extract metadata and permissions from APK files.	Analysis
2. Perform static analysis for SSL/TLS misconfigurations.	Analysis

3. Detect cryptographic API misuse.	Analysis
4. Perform manual lifecycle tracing for sensitive data leakage.	Analysis
5. Aggregate static and manual analysis results.	Evaluation
6. Categorize vulnerabilities using CWE standards.	Evaluation
7. Generate detailed vulnerability reports.	Evaluation
8. Provide an interactive dashboard for result visualization.	Evaluation
9. Allow users to upload APK files for analysis.	Evaluation
10. Minimize false positives and negatives in findings.	Analysis & Evaluation

Table 10: Mapping of Components with Requirements

6. 5 Elaborating Deployment Diagram

First, a deployment diagram is built. Then at a later phase, it is refined into instance form - that is, instantiating each deployment unit with created artifacts etc. According to UML2.0 Deployment specification and Pressman(Page 248-249). I refine the components into the following instance form for deployment.

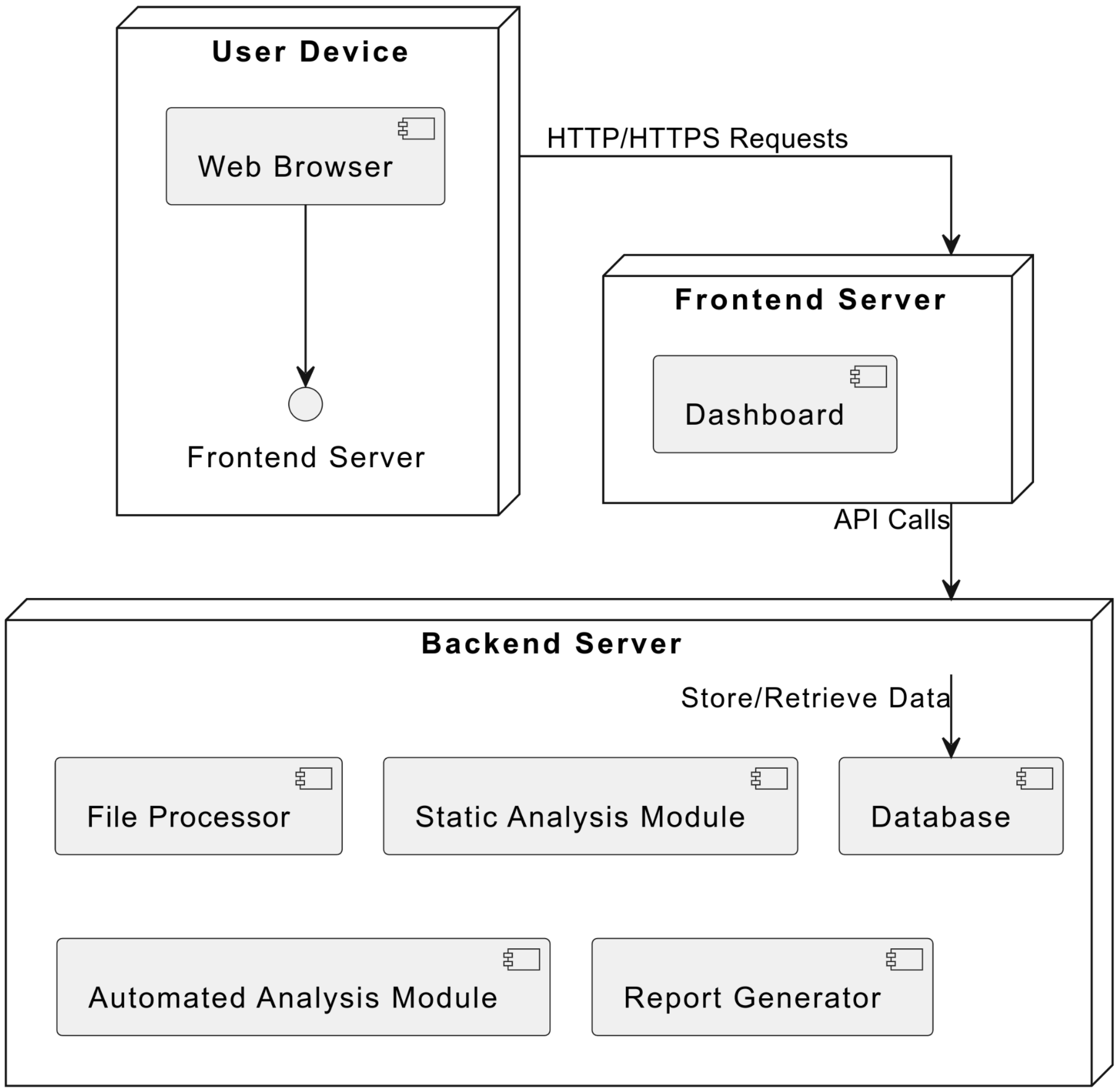


Figure-12: Instantiating Components into Artifacts and Deploying them

7. User Interface & User Manual

MoSec is a specialized security analysis platform designed to evaluate the security of branchless banking applications, particularly those distributed as APK files. With the increasing reliance on mobile banking, ensuring the security of financial applications has become a critical concern. Many mobile banking apps suffer from vulnerabilities such as weak cryptographic implementations, improper SSL/TLS configurations, and potential data leakage. MoSec aims to identify and mitigate these risks by providing a comprehensive and automated security assessment.

The platform begins by allowing users to upload APK files through a web interface, where the backend utilizes Apktool to extract essential metadata, permissions, and the application manifest. This provides an initial overview of the app's structure and functionality. For deeper security insights, MoSec employs Mallodroid for static analysis, detecting SSL/TLS vulnerabilities, improper cryptographic implementations, and access control flaws. It also integrates the Qualys SSL Server Test to assess remote SSL/TLS endpoints referenced within the APK, ensuring that server-side configurations adhere to security best practices.

To further enhance security assessment, MoSec automates manual analysis tasks. By using Baksmali for Dalvik bytecode disassembly, it identifies critical cryptographic and networking libraries, while regex-based detection mechanisms help flag risky third-party libraries. The platform also performs control flow analysis, tracing an application's lifecycle from initialization to key processes like user authentication and financial transactions, focusing on how sensitive data is handled. Reverse engineering plays a crucial role in the analysis, with MoSec utilizing the JEB Decompiler to inspect an app's code for weaknesses related to cryptography, certificate validation, and data leakage. By analyzing how authentication, session management, and encryption mechanisms are implemented, MoSec can uncover vulnerabilities that might otherwise go unnoticed.

Beyond reverse engineering, MoSec also specializes in detecting sensitive data leakage. It identifies whether financial information, personal details, or authentication tokens are being exposed through insecure communication channels or logging mechanisms. The final stage of the analysis involves compiling all findings into a dashboard, where users can view detected vulnerabilities categorized based on the Common Weakness Enumeration (CWE) classification. The platform also ensures that results undergo additional validation checks, including dynamic testing and control flow analysis, to reduce false positives and negatives. A comprehensive security report is generated, summarizing all identified vulnerabilities and providing actionable insights for developers and security teams.

By integrating multiple layers of analysis—static, dynamic, and manual—MoSec offers a robust evaluation of an application's security posture. Its goal is to help financial institutions, developers, and security researchers strengthen the security of branchless banking applications, ultimately ensuring safer and more reliable financial transactions for users.

7.1 Homepage

Homepage is the initial page of MoSec.

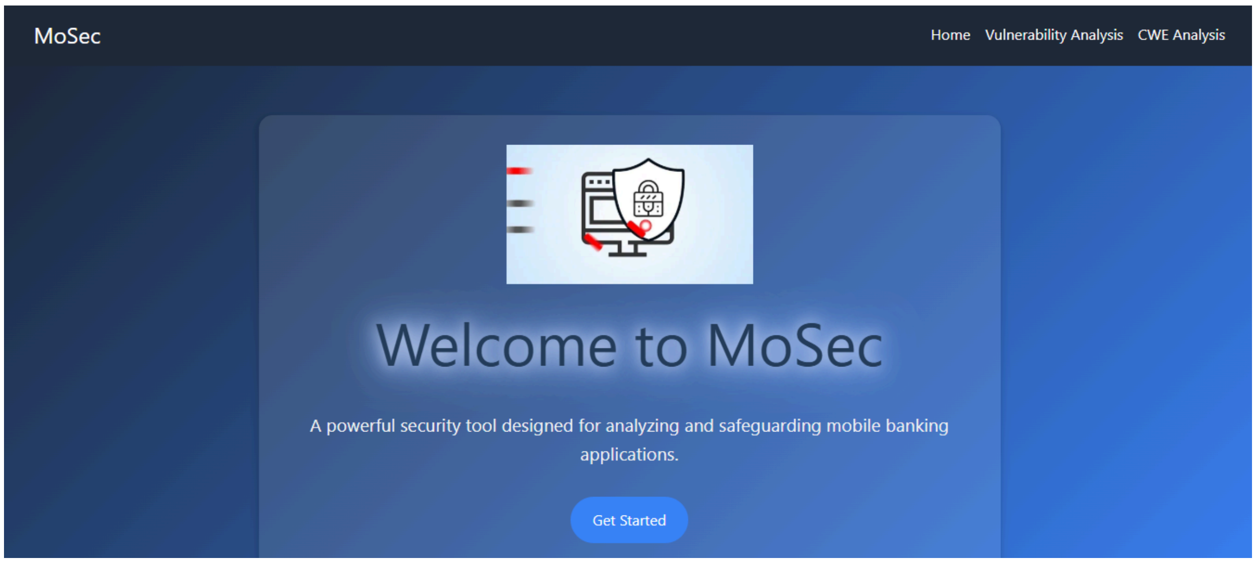
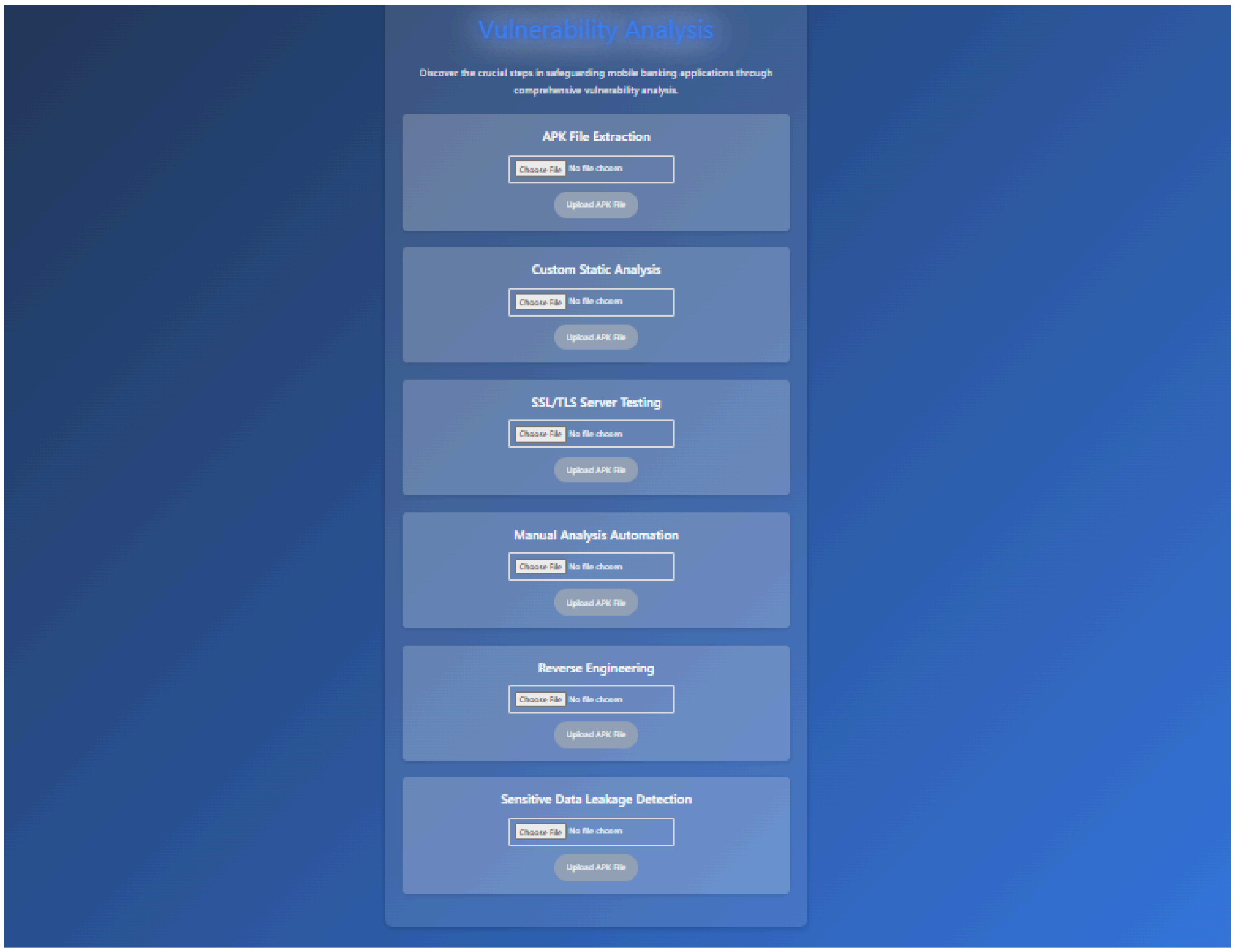

MoSec
Home
Vulnerability Analysis
CWE Analysis
Welcome to MoSec
A powerful security tool designed for analyzing and safeguarding mobile banking applications.
Get Started

Figure-13: Homepage

After Clicking Get Started button it navigates to the vulnerability analysis page.

7.2 Vulnerability Analysis

This is the page for vulnerability analysis.



Discover the crucial steps in safeguarding mobile banking applications through comprehensive vulnerability analysis.

Figure-14: Vulnerability Analysis Page

The tool will integrate the complete methodology from the referenced paper, including both automated static analysis and detailed manual teardown. The pipeline will handle APK files as input and provide outputs in the form of visual dashboards and exportable reports.

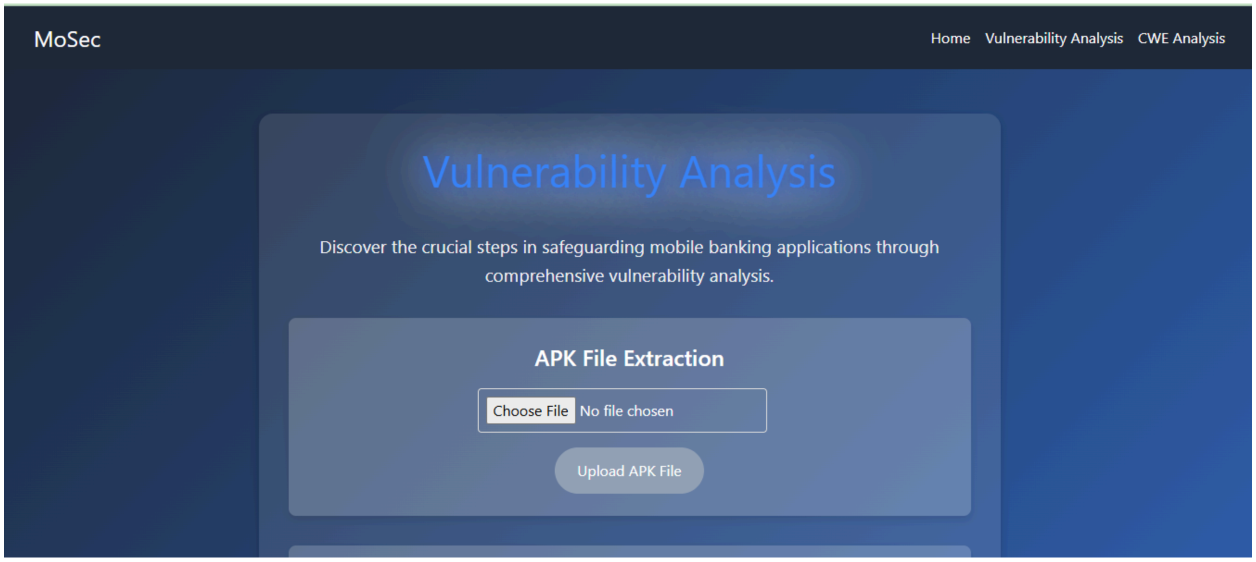


Figure-15: APK File Extraction

The user will upload APK files via the web interface. The backend will use apktool to extract the application manifest, permissions, and metadata to provide a high-level overview of the app’s structure and functionality.

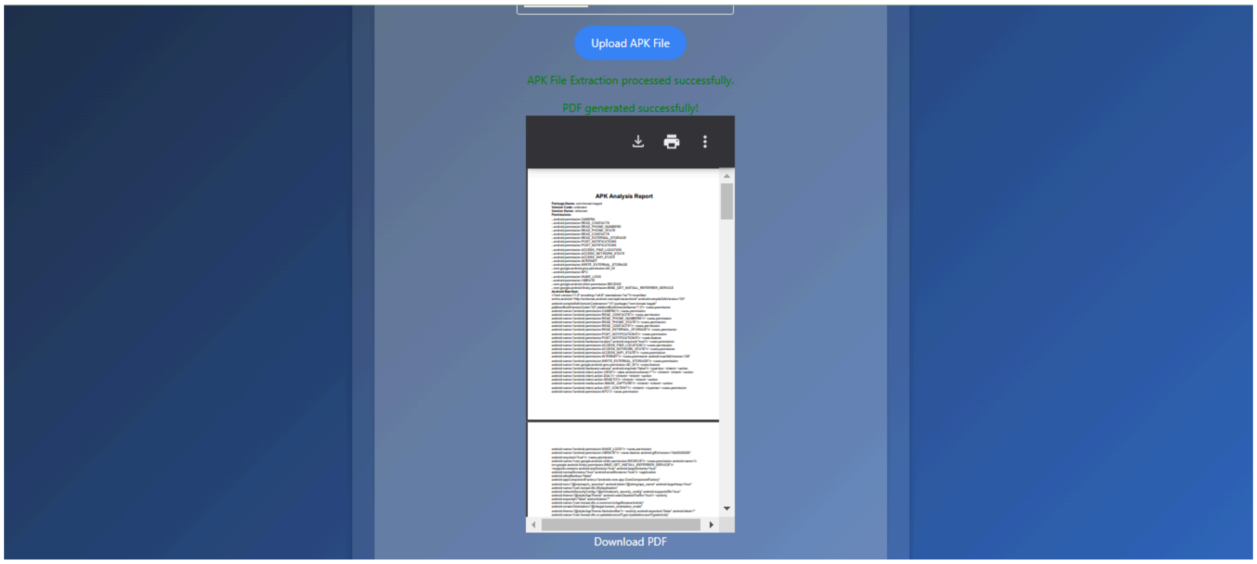

Upload APK File
APK File Extraction processed successfully.
PDF generated successfully!
APK Analysis Report
Download PDF

Figure-16: Generated Report After Uploading an APK File

The result can be downloaded as a pdf.

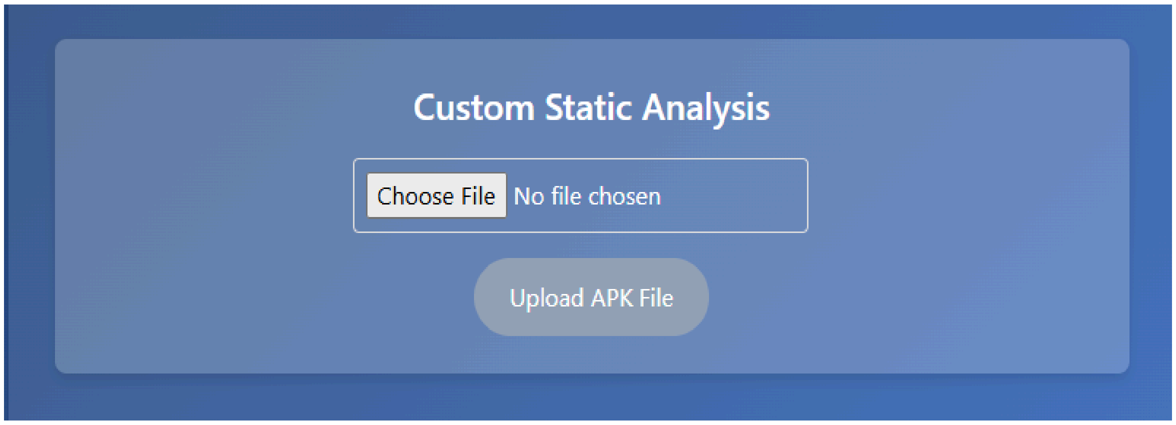
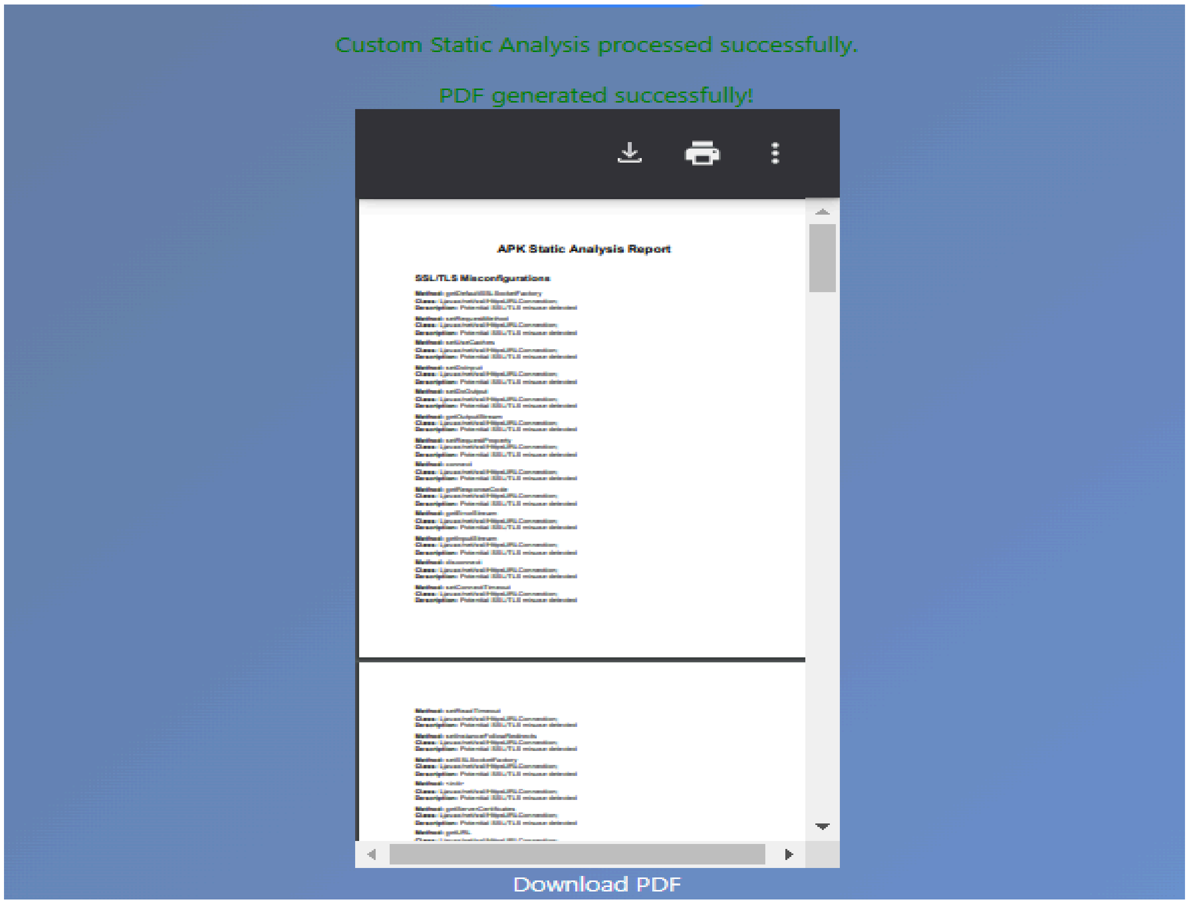


Figure-17: Custom Static Analysis

Custom static analysis tool using the Androguard API to detect SSL/TLS misconfigurations and cryptographic API misuse. Analyze the SSL/TLS certificate validation, improper cryptography, and access control issues. Provide a baseline of common SSL/TLS weaknesses, such as weak encryption, unvalidated certificates, or insecure key exchanges.


Custom Static Analysis processed successfully.
PDF generated successfully!
Download PDF

APK Static Analysis Report

SSL/TLS Misconfigurations

Method: getDefaultSSLSocketFactory
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: setRequestMethod
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: setUseCaches
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: setDoInput
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: setDoOutput
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: getOutputStream
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: setRequestProperty
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: connect
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: getResponseCode
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: getErrorStream
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: getInputStream
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: disconnect
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: setConnectTimeout
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: setReadTimeout
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: setInstanceFollowRedirects
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: setSSLSocketFactory
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: <init>
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: getServerCertificates
Class: Ljavax/net/ssl/HttpsURLConnection;
Description: Potential SSL/TLS misuse detected

Method: getURL
Class: [illegible]

Figure-18: Generated Report After Custom Static Analysis

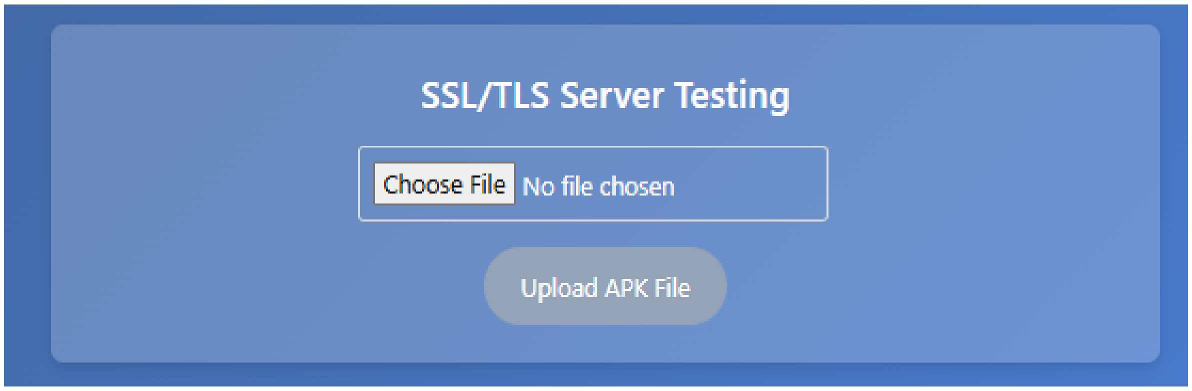
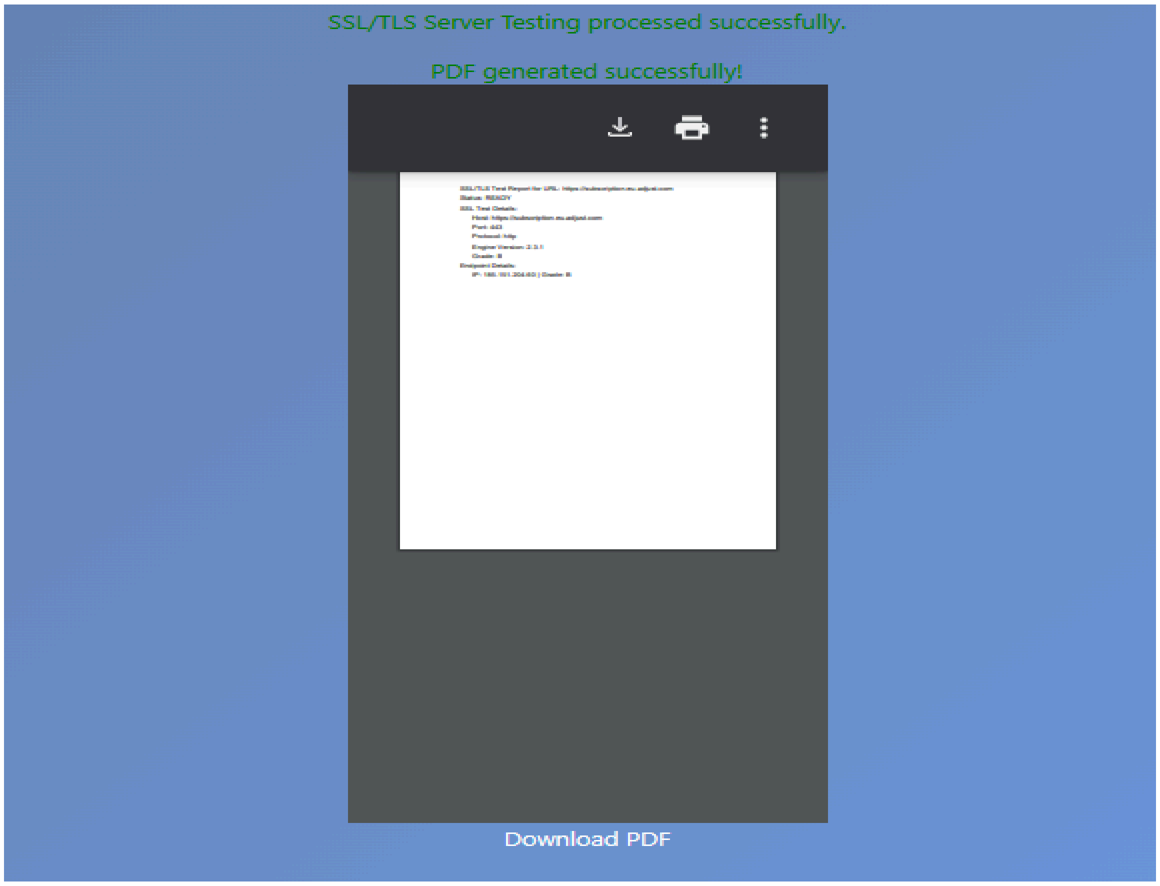


Figure-19: SSL/TLS Server Testing

Using Qualys SSL Server Test to assess the security of remote SSL/TLS endpoints referenced in the APK. The test will examine the server-side configuration for protocol strength, cipher choices, and certificate validity.


SSL/TLS Server Testing processed successfully.

PDF generated successfully!

```
SSL/TLS Test Report for URL: https://subscription.eu.adjust.com
Status: READY
SSL Test Details:
    Host: https://subscription.eu.adjust.com
    Port: 443
    Protocol: http
    Engine Version: 2.3.1
    Grade: B
Endpoint Details:
    IP: 185.151.204.60 | Grade: B
```

Figure-20: Generated Report After SSL/TLS Server Testing

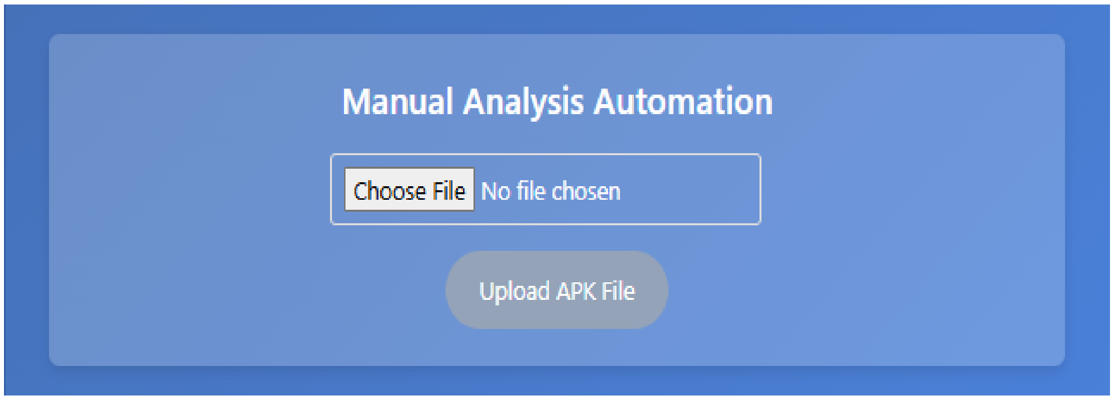
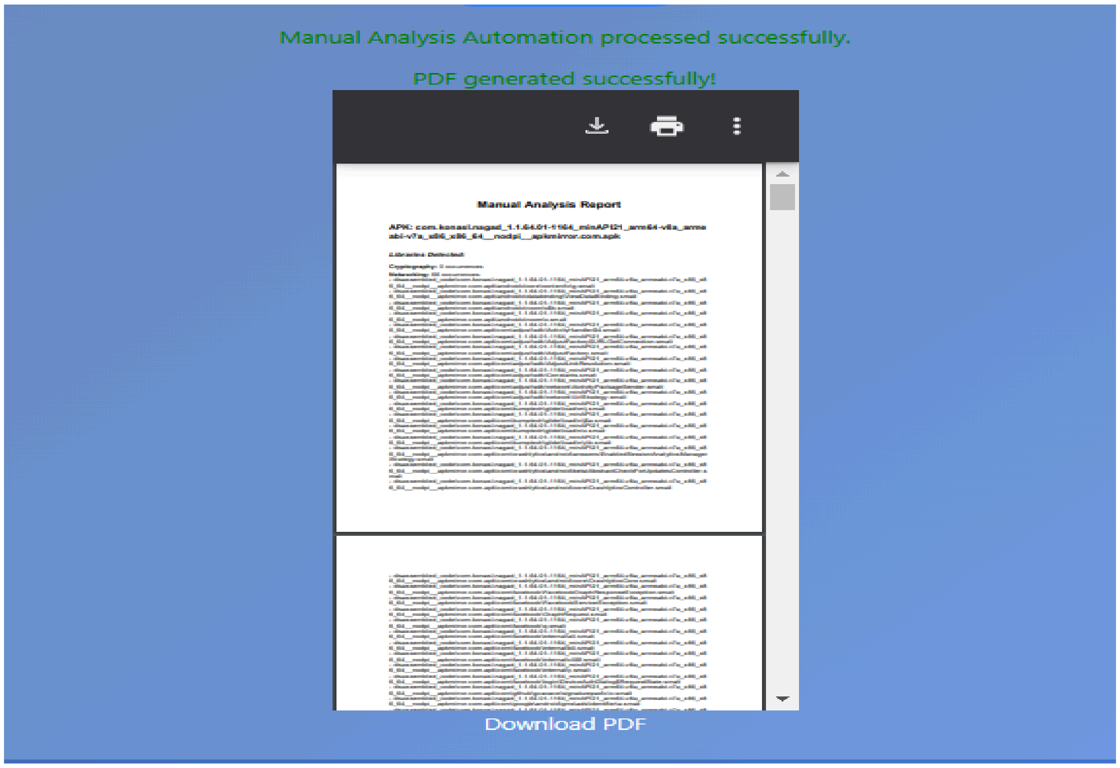


Figure-21: Manual Analysis Automation

○ Dalvik Bytecode Disassembly: Use Baksmali to disassemble the APK's bytecode and identify critical libraries such as cryptographic and networking modules.

○ Library Detection: Automate the detection of cryptography libraries and dangerous third-party libraries using regex and pattern matching scripts.

○ Lifecycle Tracing: Perform control flow analysis of the app, from the onCreate() method to registration, login, and money transfer functionalities, focusing on sensitive data handling.



Manual Analysis Automation processed successfully.

PDF generated successfully!

Manual Analysis Report

APK: [illegible]

Libraries Detected:

[illegible]

Figure-22: Generated Report After Manual Analysis Automation

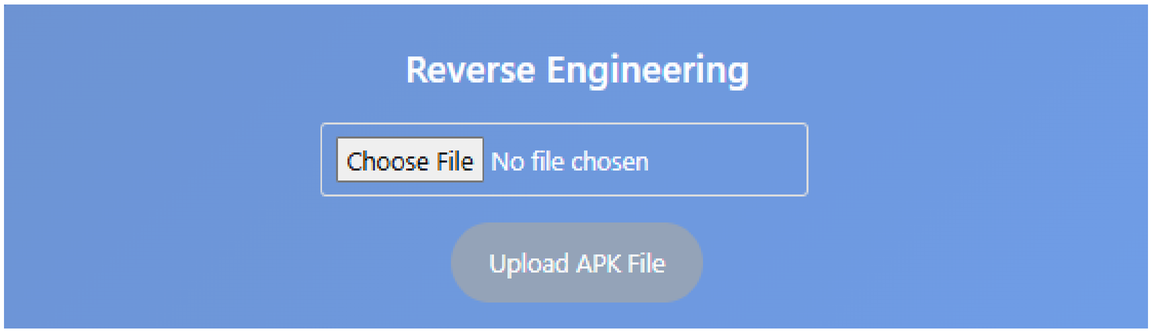
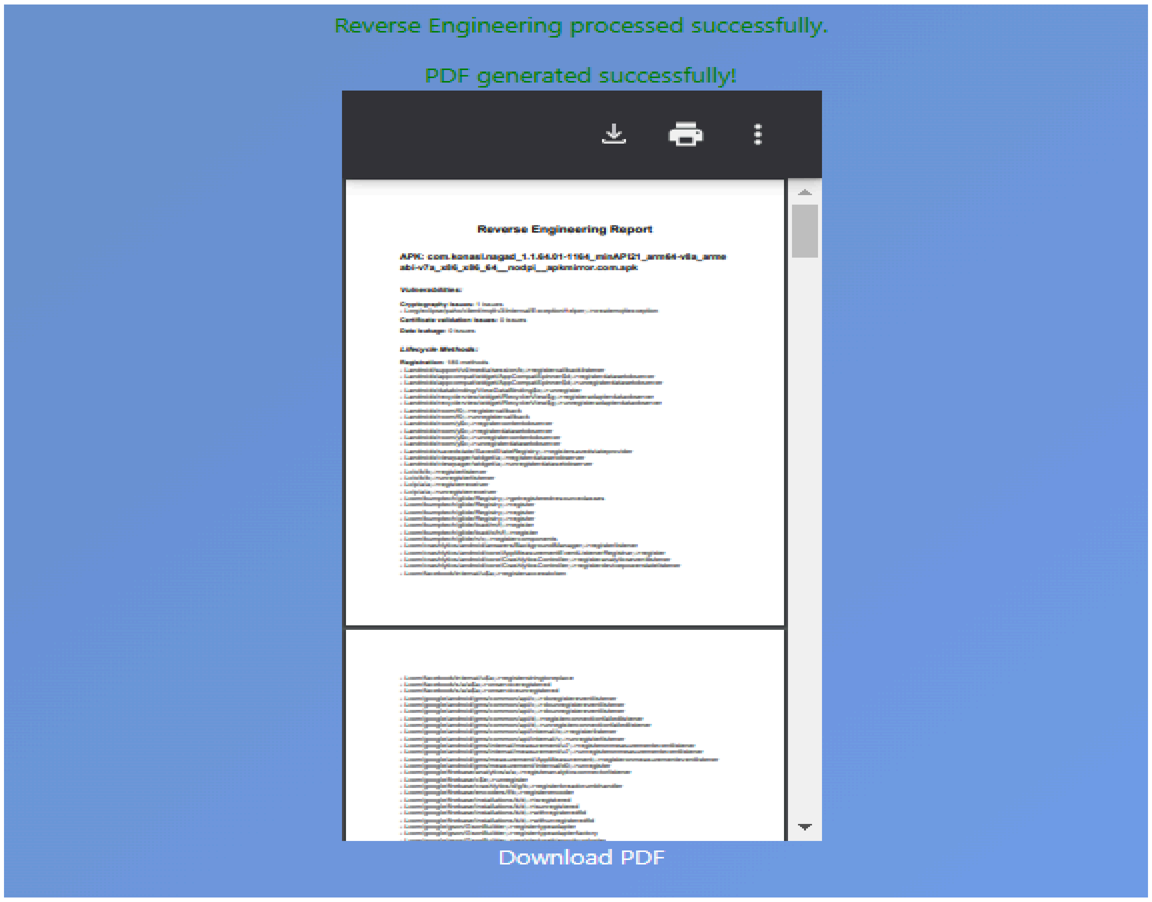


Figure-23: Reverse Engineering

○ Use the JEB Decompiler to reverse-engineer the app and inspect its code. This phase will focus on verifying vulnerabilities related to cryptography, certificate validation, and data leakage.

○ Trace the entire application lifecycle, starting with registration and continuing through user authentication, session management, and transaction processes.

○ Verify the presence of vulnerabilities in sensitive functionalities, including authentication, cryptographic procedures, and data handling.


Reverse Engineering processed successfully.

PDF generated successfully!

Reverse Engineering Report

APK: com.konasl.nagad_1.1.64.01-1164_minAPI21_arm64-v8a_armeabi-v7a_x86_x86_64__nodpi__apkmirror.com.apk

Vulnerabilities:

Cryptography Issues: 1 issues
[illegible]

Certificate validation issues: 0 issues

Data leakage: 0 issues

Lifecycle Methods:

Registration: [illegible] methods
[illegible]

Figure-24: Generated Report After Reverse Engineering

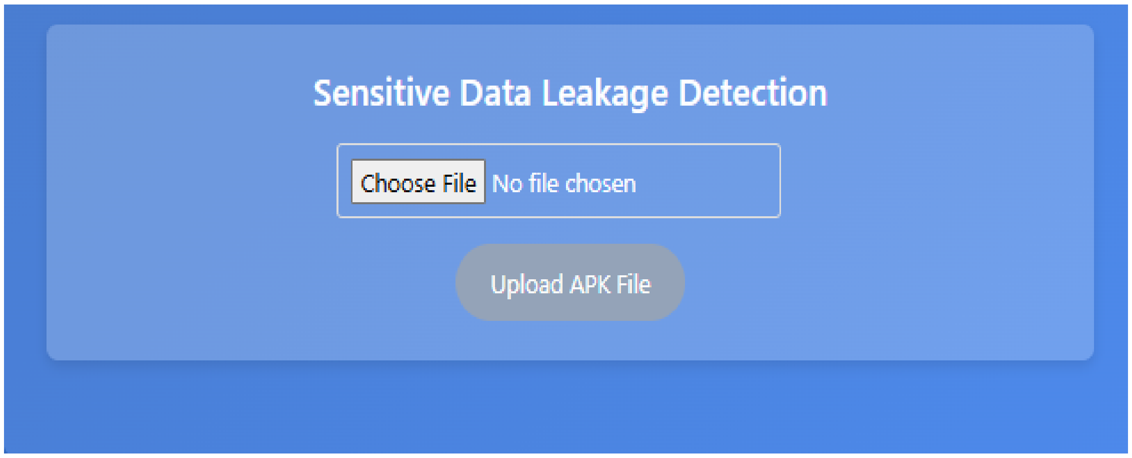
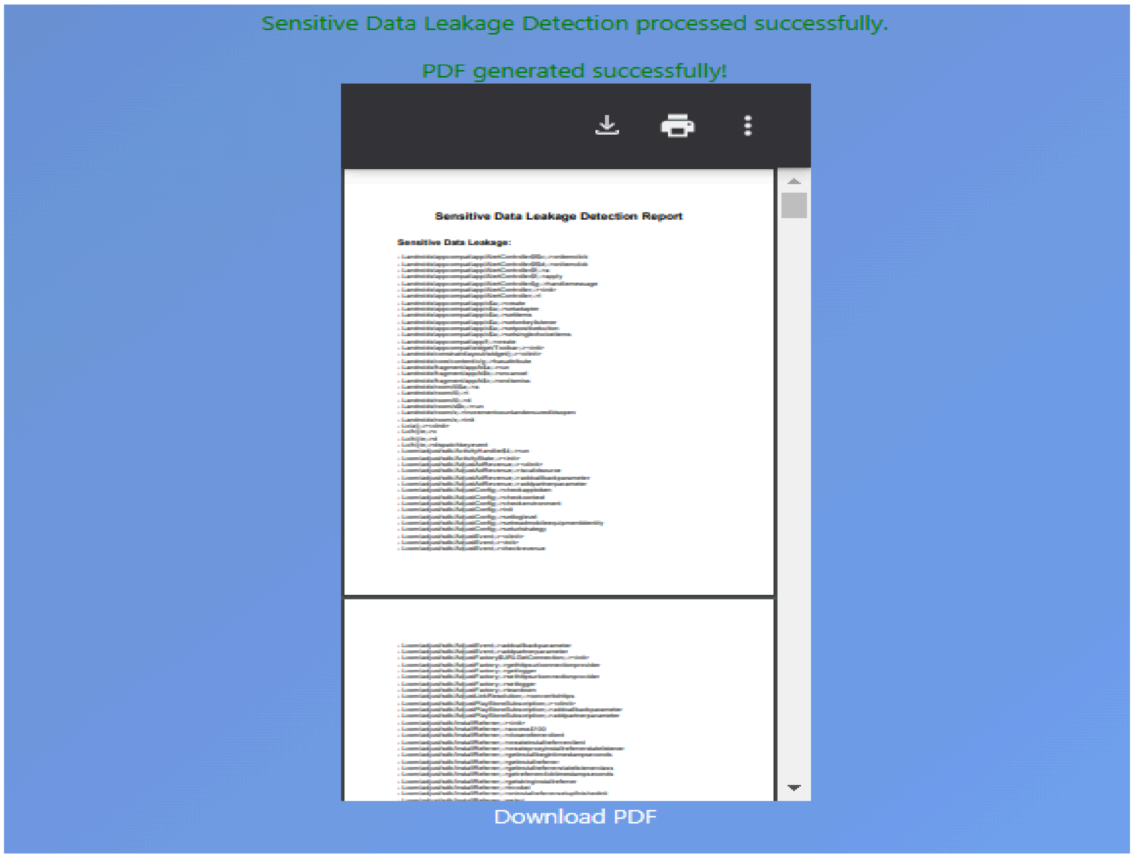


Figure-25: Sensitive Data Leakage Detection

Analyze whether sensitive data, such as financial records or personal information is leaked through insecure communication channels or logging.



Sensitive Data Leakage Detection processed successfully.

PDF generated successfully!

Sensitive Data Leakage Detection Report

Sensitive Data Leakage:

[illegible]

Figure-26: Generated Report After Sensitive Data Leakage Detection

7.3 CWE Analysis

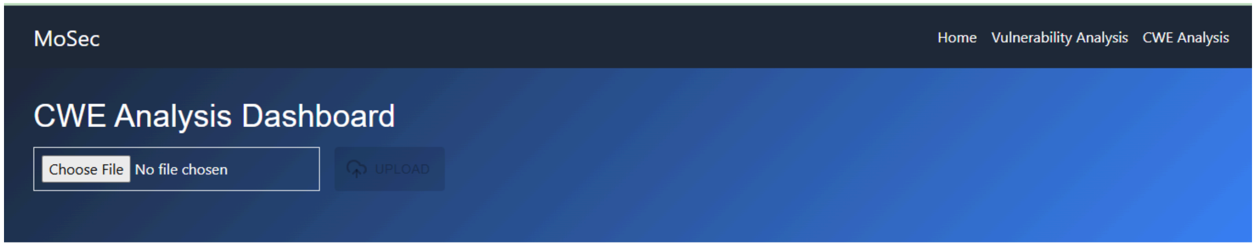
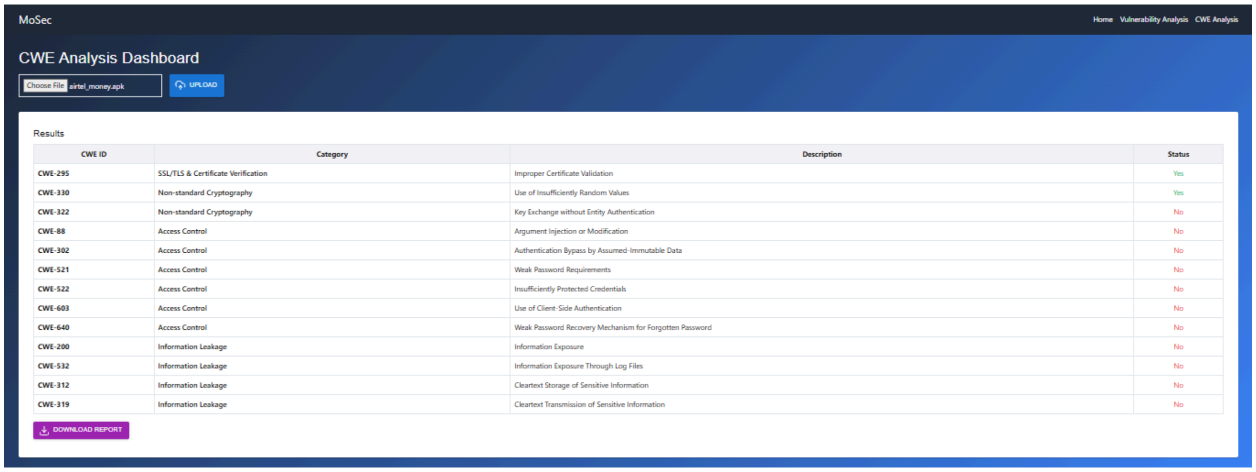


Figure-27: CWE Analysis

The results of both the static and manual analysis will be displayed on a dashboard. This includes identified vulnerabilities such as SSL/TLS protocol errors, cryptographic weaknesses, data leakage, and authentication flaws. Generate a comprehensive report for users to download, summarizing the analysis with details on each detected vulnerability, categorized using CWE (Common Weakness Enumeration) classifications.



Results

CWE ID	Category	Description	Status
CWE-295	SSL/TLS & Certificate Verification	Improper Certificate Validation	Yes
CWE-330	Non-standard Cryptography	Use of Insufficiently Random Values	Yes
CWE-322	Non-standard Cryptography	Key Exchange without Entity Authentication	No
CWE-88	Access Control	Argument Injection or Modification	No
CWE-302	Access Control	Authentication Bypass by Assumed-Immutable Data	No
CWE-521	Access Control	Weak Password Requirements	No
CWE-522	Access Control	Insufficiently Protected Credentials	No
CWE-603	Access Control	Use of Client-Side Authentication	No
CWE-640	Access Control	Weak Password Recovery Mechanism for Forgotten Password	No
CWE-200	Information Leakage	Information Exposure	No
CWE-532	Information Leakage	Information Exposure Through Log Files	No
CWE-312	Information Leakage	Cleartext Storage of Sensitive Information	No
CWE-319	Information Leakage	Cleartext Transmission of Sensitive Information	No

Figure-28: CWE Analysis Dashboard

8. Preliminary Test Plan

In this chapter, a high-level description of testing goals and a summary of features to be tested are presented.

8.1 High-Level Description of Testing Goals

The testing goals for MoSec are as follows.

Testing Goals:

1. **Verify Functional Requirements**: Ensure all core functionalities, such as file uploads, metadata extraction, static and manual analysis, and report generation, work as expected.
2. **Validate Security Analysis Accuracy**: Confirm that the static and manual analysis modules accurately identify vulnerabilities (e.g., SSL/TLS misconfigurations, cryptographic misuse) with minimal false positives and negatives.
3. **Ensure Data Integrity**: Verify that all stored and retrieved data in the database remain consistent, secure, and accurate throughout the analysis and reporting lifecycle.
4. **Test API Performance**: Assess the responsiveness and reliability of APIs connecting the frontend and backend components under varying loads.
5. **Check Frontend Usability**: Ensure the dashboard is user-friendly, responsive, and provides clear and actionable insights into the analysis results.
6. **Assess Scalability**: Test the system's ability to handle large numbers of concurrent users and high-volume APK submissions without performance degradation.
7. **Validate Security Measures**: Ensure secure handling of APK files, user data, and vulnerability reports, including encryption of sensitive data and secure communication protocols (e.g., HTTPS).
8. **Perform End-to-End Testing**: Simulate complete workflows from APK upload to final report generation to ensure seamless integration across all components.

9. **Test Cross-Platform Compatibility**: Verify that the web application functions correctly across different browsers and devices.
10. **Evaluate Error Handling**: Confirm that the system gracefully handles invalid APKs, failed analyses, and other unexpected errors without crashing.

8.2 Test Cases

Test Case 1: Upload Valid APK

- **Description**: Verify that a valid APK file can be successfully uploaded.
- **Preconditions**: User has a valid APK file ready for upload.
- **Test Steps**:
    - Open the dashboard in a web browser.
    - Navigate to the "Upload APK" section.
    - Select a valid APK file and click "Upload."
- **Expected Outcome**:
    - The file is uploaded successfully.
    - A confirmation message is displayed.
    - The system starts processing the APK

Result: Passed

Test Case 2: Upload Invalid File

- **Description**: Verify that non-APK files cannot be uploaded.
- **Preconditions**: User has an invalid file (e.g., .txt or .jpg).
- **Test Steps**:
    - Open the dashboard in a web browser.
    - Navigate to the "Upload APK" section.
    - Select an invalid file and click "Upload."
- **Expected Outcome**:
    - The upload is rejected.

    - An error message indicates that only APK files are allowed.

Result: Passed

Test Case 3: Extract APK Metadata

- **Description**: Validate that the system correctly extracts metadata from the APK.
- **Preconditions**: A valid APK file has been uploaded.
- **Test Steps**:
    - Upload a valid APK file.
    - Wait for the system to process the file.
    - View the extracted metadata in the dashboard.
- **Expected Outcome**:
    - Metadata (e.g., package name, version, permissions) is displayed accurately.

Result: Passed

Test Case 4: Perform Static Analysis

- **Description**: Verify that the static analysis module detects vulnerabilities in the APK.
- **Preconditions**: A valid APK file has been uploaded.
- **Test Steps**:
    - Upload a valid APK file.
    - Wait for static analysis to complete.
    - View the analysis results on the dashboard.
- **Expected Outcome**:
    - Detected vulnerabilities (e.g., SSL/TLS misconfigurations) are listed.

Result: Passed

Test Case 5: Perform Manual Analysis

- **Description**: Verify that the manual analysis module identifies sensitive data leaks.
- **Preconditions**: A valid APK file has been uploaded.

- **Test Steps**:
  - Upload a valid APK file.
  - Wait for manual analysis to complete.
  - View the manual analysis findings on the dashboard.
- **Expected Outcome**:
  - Sensitive data leaks (e.g., unencrypted logs) are listed.

Result: Passed

Test Case 6: Aggregate Analysis Results

- **Description**: Validate that the system combines static and manual analysis results accurately.
- **Preconditions**: Static and manual analyses are completed for an APK.
- **Test Steps**:
  - Perform both static and manual analyses.
  - View the aggregated results on the dashboard.
- **Expected Outcome**:
  - Results are displayed as a unified report.

Result: Passed

Test Case 7: Generate Detailed Report

- **Description**: Verify that the system generates a downloadable report.
- **Preconditions**: Analysis (static and manual) has been completed for an APK.
- **Test Steps**:
  - Complete the analysis of an APK.
  - Click the "Download Report" button on the dashboard.
- **Expected Outcome**:
  - A detailed report in PDF or another supported format is downloaded.

Result: Passed

Test Case 8: Handle Large APK File

- **Description**: Verify that the system can process large APK files.
- **Preconditions**: User has a large APK file (e.g., 100 MB or more).
- **Test Steps**:
  - Upload a large APK file.
  - Wait for the system to process the file.
- **Expected Outcome**:
  - The file is processed successfully without performance issues.

Result: Passed

Test Case 9: Prevent Data Loss

- **Description**: Verify that no data is lost during analysis.
- **Preconditions**: An APK is uploaded, and the database is functional.
- **Test Steps**:
  - Upload an APK and initiate analysis.
  - Check the database for stored results.
  - View results on the dashboard.
- **Expected Outcome**:
  - Data stored in the database matches what is displayed on the dashboard.

Result: Passed

Test Case 10: Ensure Secure Communication

- **Description**: Validate that all communications use HTTPS.
- **Preconditions**: The system is deployed in a live environment.
- **Test Steps**:
  - Monitor network traffic while interacting with the dashboard.
  - Check for insecure communication protocols.
- **Expected Outcome**:

    - All traffic between the frontend and backend is encrypted using HTTPS.

Result: Passed

Test Case 11: Display Dashboard Responsiveness

- **Description**: Verify that the dashboard remains responsive under heavy load.
- **Preconditions**: Multiple users are trying to access the dashboard simultaneously.
- **Test Steps**:
    - Simulate 50+ concurrent users accessing the dashboard.
    - Monitor response times and usability.
- **Expected Outcome**:
    - The dashboard remains functional and responsive.

Result: Passed

Test Case 12: Handle Invalid APK Structure

Description: Verify that corrupted APKs are handled gracefully.

Preconditions: User has a corrupted APK file.

Test Steps:

- Upload a corrupted APK file.
- Wait for the system to process the file.

Expected Outcome:

- The upload is rejected, and an error message indicates the issue.

Result: Pending

Test Case 13: Security testing

Descriptions: Detection of malicious apk.

Precondition: User has been registered.

Test Steps:

- Upload a malicious apk.

Expected Result:

- The system generates a report similar to “Invalid apk”.

Result: Passed

Test Case 14: Load balance testing

Description: Simulating a high volume of requests to evaluate its performance under load.

Precondition: Multiple users are trying to access the dashboard simultaneously

Test Steps: Run the test script.

Expected Result: Response time should be between 200-300ms. Throughput value should be between 150 to 200 users.

Result: Pending

9. Conclusion

The **MoSec** Framework represents a pivotal advancement in the security analysis of mobile money applications, particularly addressing the unique challenges of these applications in developing regions. By combining automated static analysis with manual techniques, MoSec offers a comprehensive approach to identifying vulnerabilities such as cryptographic misconfigurations, insecure SSL implementations, and data leakage. This dual-layered strategy ensures both efficiency and accuracy, providing actionable insights to developers and stakeholders to enhance the security posture of mobile financial ecosystems.

A key strength of MoSec lies in its adaptability and scalability. The framework's modular architecture allows it to integrate seamlessly with various analysis tools and methodologies, ensuring relevance in an ever-evolving threat landscape. By presenting results through an interactive dashboard and detailed reports, MoSec not only identifies security flaws but also simplifies the remediation process for its users. The use of standardized classifications like CWE ensures consistency and transparency in reporting, making it easier for developers to prioritize and address critical issues.

Overall, the MoSec Framework is more than a security analysis tool—it is a proactive solution aimed at fostering trust in mobile financial systems. By empowering developers with the ability to detect and mitigate vulnerabilities early, MoSec helps reduce risks for millions of users who depend on these applications daily. As mobile money continues to expand its reach, the contribution of projects like MoSec becomes increasingly vital to ensure secure and reliable access to financial services globally.

References

[1] C. Penicaud and A. Katakam. Mobile Financial Services for the Unbanked: State of the Industry 2013. Technical report, GSMA, Feb. 2014.

[2] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In Proc. 20th USENIX Security Sym., San Francisco, CA, USA, 2011.

[3] P. Traynor, P. McDaniel, and T. La Porta. Security for Telecommunications Networks. Springer, 2008.

[4] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R.B. Butler. Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World. In Proc. 24th USENIX Security Sym., Washington, D.C, USA, 2015.