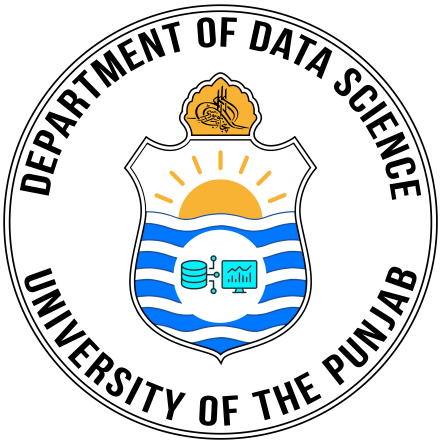
Final Year Project Proposal

SherlockDroid

GenAI-Powered Framework for Mobile App Vulnerability Analysis and Exploitation



By

Ayesha Khan BSDSF22M029

Usman Ahmed BSDSF22M014

Fatima Mubasher BSDSF22M044

Abeer Zafar BSDSF22M045

Under the supervision of

Maj (Retd) Dr. Muhammad Arif Butt

Bachelor of Science in Data Science (2022-2026)

FACULTY OF COMPUTING & INFORMATION TECHNOLOGY

(FCIT),

UNIVERSITY OF THE PUNJAB, LAHORE.

SherlockDroid

GenAI-Powered Framework for Mobile App Vulnerability Analysis and Exploitation

A project proposal presented to

University of the Punjab, Lahore

In partial fulfillment of the requirement for the degree of

Bachelor of Science in Data Science (2022-2026)

By

Ayesha Khan	BSDSF22M029
Usman Ahmed	BSDSF22M014
Fatima Mubasher	BSDSF22M044
Abeer Zafar	BSDSF22M045

FACULTY OF COMPUTING INFORMATION TECHNOLOGY (FCIT),

UNIVERSITY OF THE PUNJAB, LAHORE

Executive Summary

There is a surge in deployment of Android applications, which is accompanied by a growing attack surface, marking mobile security as a high-priority research topic. This project proposes **SherlockDroid** is an intelligent, modular framework designed to automate the process of mobile application vulnerability detection through integrated static and dynamic analysis pipelines. The system architecture incorporates reverse engineering to extract application behaviors, permissions, and potential attack vectors from uploaded APK files. These extracted features are then processed by a large language model (LLM) to generate comprehensive vulnerability reports and Proof-of-Concept (PoC) exploits, which are in-turn executed by SherlockDroid in a controlled environment.

The core methodology involves combining static analysis tools (e.g., MobSF[1], JADX[2]) with dynamic analysis tools (e.g., Frida[3]), while leveraging generative AI models for interpretability and exploit creation. This tool includes static and dynamic analysis of APKs, exploit scripts, and interactive guidance. The proposed system will be evaluated using benchmark vulnerable apps and real-world APKs like DIVA[4], InsecureBankv2[5], AndroGoat[6], Damn Vulnerable Bank[7], OWASP MASTG Test Suite[8], and datasets such as Ghera to evaluate its accuracy.

The expected outcomes include accurate vulnerability detection with reduced dynamic analysis overhead, PoC exploit generation, and AI-generated standardized reports. The framework targets security professionals, developers, and researchers, promoting ethical and safe usage for educational and defensive purposes.The framework is designed for **penetration testers**, **security researchers**, and **educators**, with a strong emphasis on *safe, ethical, and offline use only.*

Contents

List of figures

List of tables

Chapter 1

INTRODUCTION

1.1 Background

Android being an open source and customizable operating system is prone to many security flaws that can occur in both, third-party and enterprise apps. Tools like MobSF[1], JADX[2] help in reverse engineering and vulnerability scanning but are unable to provide automated exploit generation and meaningful interpretation of results.

1.2 Problem Statement

We experience a crucial void when it comes to vulnerability assessment of mobile applications and understanding their real-world consequences through Poc exploit generation.Current tools and frameworks lack a complete end-to-end solution for vulnerability assessment, meaningful explanation and exploitation using context-aware automation. SherlockDroid is hence, a smart one stop solution to automate all these tasks under a unified framework. Table 1.1 highlights the primary customer segments for SherlockDroid along with their corresponding needs and estimated global market sizes.

Customer Type	**Need or Use Case**	**Estimated Market Size**
Security Analysts and Penetration Testers	Android application assessments will be made easier for professionals through this automated and unified framework.	150,000+ professionals globally
Cybersecurity Students and Researchers	Understanding the outcomes or results of a vulnerability scan or exploitation is made easier with this tool.	500,000+ students and early researchers
Android Developers	During the development life cycle, developers can perform active analysis of their code using this automated framework.	1.5+ million globally
AI-Security Researchers	This tool can serve as a foundation or case study for further advancements.	50,000+ active researchers

Table 1.1. Target Customers and Their Estimated Market Sizes

1.3 Project Goals & Objectives

This framework aims to perform both static and dynamic analysis techniques to automate the process of identifying vulnerabilities or security flaws in Android APKs, integrate AI-based proof-of-concept exploit code, execute generated exploits in a controlled lab environment (physical Android devices), create a dashboard that allows to remotely trigger actions like accessing files, camera,contacts, microphone, etc., analyze malicious APKs, generate AI-powered comprehensive reports and bridge red team and blue team use cases. Figure 1.1 illustrates the complete workflow of how SherlockDroid achieves end-to-end vulnerability exploitation and reporting in Android applications. The following are the main objectives of this project.

- **Develop an APK Analysis Module** that implements static and dynamic analysis, automating the process of identifying vulnerabilities and malicious behaviors in Android APKs.
- **Integrate AI-Based Exploit Generation** to generate proof-of-concept (PoC) exploit code snippets for the discovered vulnerabilities.
- **Design a Real Device Exploit Framework** that enables execution of generated PoCs on physical Android devices inside the controlled testing environment.
- **Build a Remote Command and Control Interface (C2)** for remote control of compromised devices and execution of commands to attack these devices, in order to get access to microphone, contacts, etc.
- **Implement Malware Classification** using agentic AI techniques to autonomously detect and classify vulnerable apps based on the APKs uploaded.
- **Simulate Persistence Mechanisms** demonstrate techniques used to maintain access on compromised devices.
- **Generate Standardized Security Reports** combining vulnerability findings, PoC exploit impact analysis, etc.

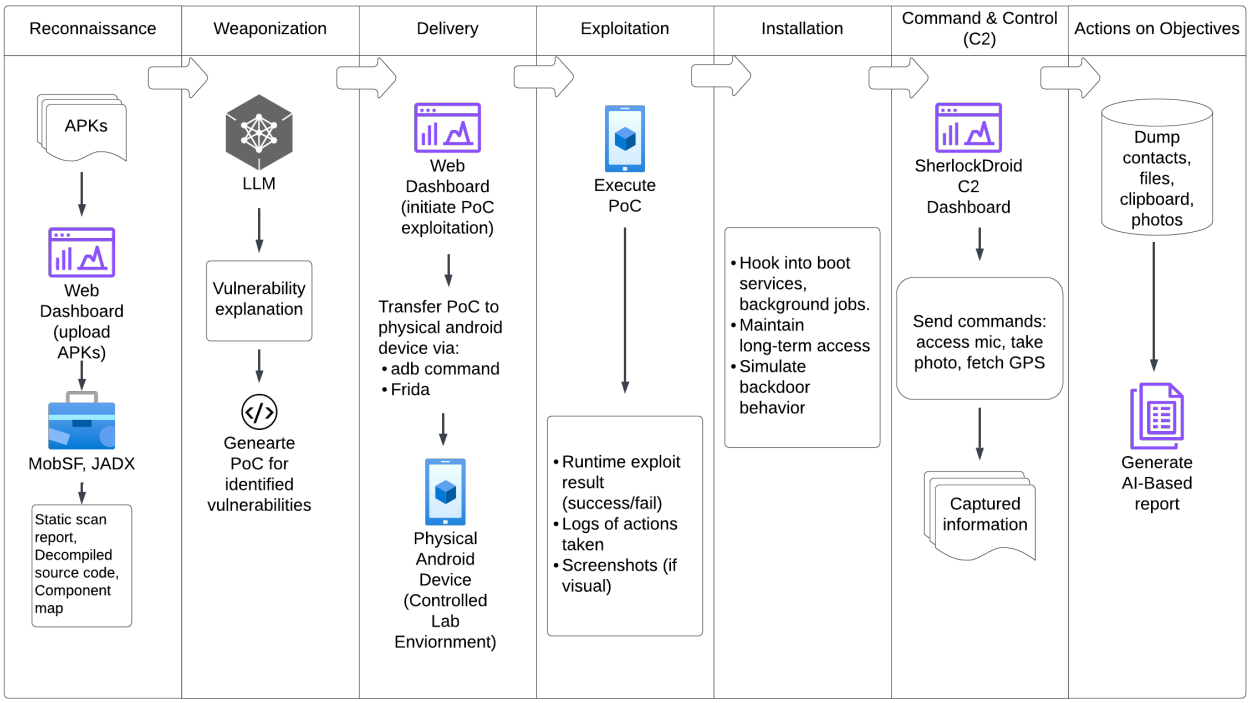


Fig. 1.1. Kill Chain architecture that illustrates the complete offensive security workflow facilitated by SherlockDroid.

1.4 Scope

The project aims to design and implement SherlockDroid, a unified, modular and intelligent framework that combines static as well as dynamic analysis with Gen-AI that automates the process of vulnerability assessment and exploitation for Android applications. The framework will include sub-modular design to optimize dynamic analysis that will in turn reduce overall execution time, automated reporting system and tuning LLMs to generate exploits and human readable interpretations of analysis report. An interactive LLM agent focused on apk structure for explaining the results. It also includes demoing successful device takeovers, AI-assisted PoC generation, and persistent malware behavior under controlled conditions.Moreover, it uses LLMs (GPT-4, VulnGPT) to explain vulnerabilities, guide users, and generate PoC scripts and remediation advice.This framework will be targeted towards ethical hackers, students and android app developers. Limitations of this project include iOS application analysis, zero-day vulnerability detection, exploitation beyond proof of concept (PoC) and offensive usage.

1.5 High-level System Components

SherlockDroid consists of the following key components:

- **Static Analysis Module**: Uses tools like JADX and MobSF to analyze APKs, extract code, and identify vulnerabilities via signatures and permissions.
- **Dynamic Analysis Module**: Runs the app in a sandbox using Frida/Xposed, monitors real-time behaviors and API calls.

- **GenAI Layer**: Employs large language models to generate PoC exploits, explain findings, and guide users through a chatbot interface.
- **PoC Validation Engine**: Executes and verifies the effectiveness of generated exploits in a controlled environment.
- **Reporting Module**: Presents results in a user-friendly format with severity rankings and downloadable reports.
- **User Interface**: Offers both web and mobile interfaces for interactive analysis and control.
- **Security Ethics Layer**: Enforces responsible use policies and ensures the tool is used ethically for research and academic purposes.

1.5.1 Mobile Application

Although SherlockDroid itself does not provide a separate mobile app, its **Progressive Web App (PWA)** compatibility through the React-based front-end enables mobile-device accessibility for viewing results and interacting with the LLM assistant.

The mobile-facing objectives include:

- Allowing users (e.g., testers or students) to upload APK files directly from mobile browsers.
- Interacting with chatbot-based explanations of vulnerabilities via touch-optimized UI.
- Viewing scan summaries, exploit code, and remediation tips in a mobile-friendly dashboard.

Future iterations may include a dedicated Android app that allows remote control of the analysis pipeline or on-device scanning in limited scope.

1.5.2 Web Application

The web application acts as the **user interface layer** of SherlockDroid and is developed using **React.js** and **Material UI**. It supports PWA standards for responsive usage on both desktop and mobile browsers.

Key features of the web application include:

- User authentication and secure login system with JWT-based sessions.
- APK upload interface for static and dynamic analysis initiation.
- Dashboard to view:
  - Static analysis results (permissions, API calls, code smells)
  - Dynamic behavior logs and Frida output
  - AI-generated attack vectors, PoC exploits and natural language reports

- Embedded chatbot interface that uses LLMs to answer user questions regarding vulnerabilities, APK structure, and remediation strategies.

Table 1.2 represents the tools and technologies that will be used throughout the development of SherlockDroid.

1.5.3 Backend Services

The backend services power the core intelligence and automation within the SherlockDroid framework. They are designed using **Python Django** for robust API development and routing, and seamlessly integrate with LLMs and static/dynamic analyzers to deliver intelligent automation.

The backend handles:

- File management and validation of uploaded APKs.
- Invocation of static analysis tools like MobSF[1], JADX[2].
- Launching sandboxed environments and emulators for dynamic analysis via Frida[3] and logcat.
- Communication with AI agents for:
  - Generating PoC exploits
  - Producing natural language summaries
  - Interacting with users in chatbot sessions
- Storing and retrieving analysis results from a cloud-hosted **MongoDB Atlas** database.

For security, all backend APIs are protected via role-based JWT tokens. Backend services are containerized using Docker and orchestrated through Kubernetes for scalable, production-grade deployment.

1.6 List of Optional Functional Units

- **Dark Mode and UI Personalization:** Users can switch between modes and customize dashboard for accessibility.
- **Notifications:** Notifications or automated E-mails after completion of scans.
- **Real-Time Analysis Progress View:** Users can view real-time progress of scanning, detection, and extraction of attack vectors via LLMs, generating PoCs, and executing these in a controlled environment.
- **Interactive Visualizations:** Permission heatmaps, tracking graphs, and threat confidence levels can be visualized by users.

- **Mobile Companion App (SherlockMobile Lite):** A lightweight scanner app developed to perform scans on user's mobile phone for detecting malicious APKs.
- **Scheduled Scan Automation:** Users may set up periodic scans.

1.7 Application Architecture

SherlockDroid follows a modular **three-layer architecture** to ensure scalability, maintainability, and ease of testing. Figure 1.2 illustrates the overall architecture of SherlockDroid, highlighting its modular flow from APK submission to LLM-assisted vulnerability analysis and exploit testing. These layers include the Presentation Layer, the Business Logic Layer, and the Data Layer.

The **Presentation Layer** is responsible for user interaction. It includes a web dashboard built using **React.js** and **Material UI**, providing users with an intuitive interface to upload APK files, view vulnerability reports, and interact with an AI assistant for detailed explanations.

The **Business Logic Layer** serves as the core engine of SherlockDroid. It includes static analysis components powered by **MobSF**[1] and **JADX**[2], which scan APKs for insecure permissions, hardcoded secrets, and vulnerable libraries. The dynamic analysis component uses an Android Emulator in conjunction with **Frida** to detect runtime issues such as data leaks and insecure API calls.

The **Data Layer** is responsible for storage and persistence. It uses **MongoDB** to manage structured data such as scan results, user history, and vulnerability reports. APK files, analysis logs, and AI-generated outputs are stored in either local storage or cloud solutions like **AWS S3**.

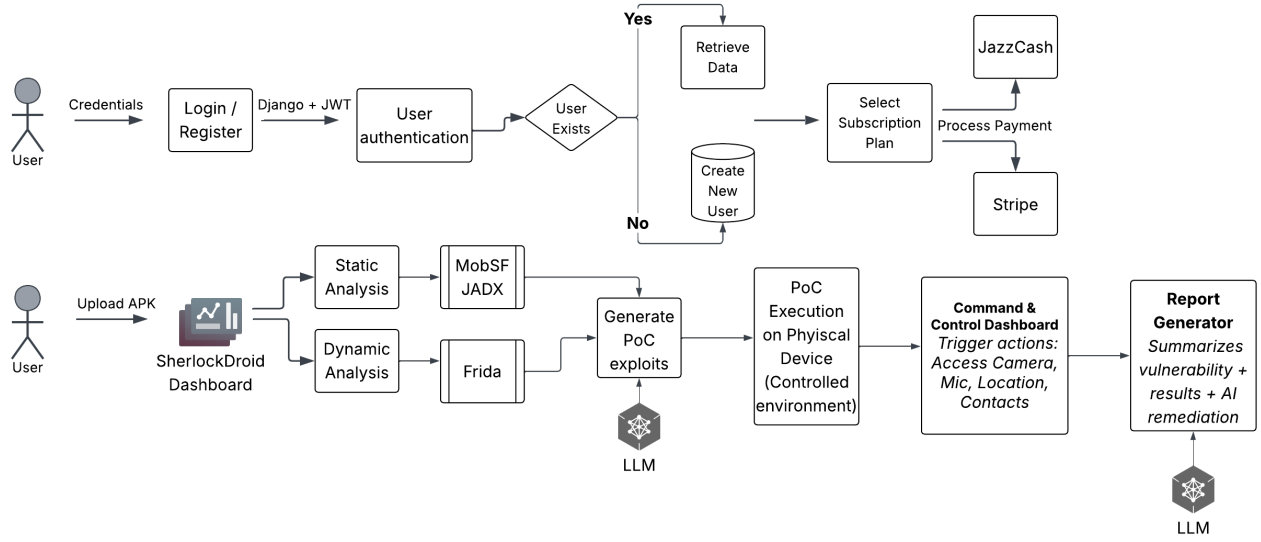


Fig. 1.2. System architecture of SherlockDroid showcasing APK analysis, PoC generation, physical device testing, and AI-powered vulnerability reporting.

1.8 System Limitations and Constraints

1.8.1 Limitations

- Continuous internet access is required for performing large language model (LLM)-based analysis and dynamic vulnerability checks; offline usage is not supported.
- Detection accuracy may vary for heavily obfuscated or encrypted APKs.

1.8.2 Constraints

- The system is focused on APK analysis only; iOS application support is not included.
- Zero-Day Vulnerability Detection is not the scope of this project.

1.9 Tools and Technologies Used with Reasoning

Layer	**Best Choice**	**Why?**
Frontend	React.js + Material UI	Modern, responsive UI; rapid development; component-based design ideal for dashboards and interactivity
Backend	Django + Django REST Framework + JWT	Python-based; secure, scalable; REST API support; JWT used for stateless authentication
Database	MongoDB (Self-hosted or Atlas)	Flexible document-based schema; fast reads/writes; stores scan results and user metadata efficiently
Static Analysis	MobSF + JADX	MobSF and JADX are trusted open-source tools for static APK inspection, permission analysis, and reverse engineering
Dynamic Analysis	Frida + Emulator	Frida enables runtime monitoring of APIs, memory, and data flows inside emulated environments
AI Layer	Python (LLM API e.g., GPT-4 or LLaMA 3 via Flask)	Python has rich AI/ML ecosystem; Flask is lightweight and easy to integrate with model APIs for vulnerability explanation
File Storage	AWS S3	Secure, scalable cloud storage for APKs, reports, and logs
Containerization	Docker + Docker Compose	Simplifies deployment; isolates tools like MobSF, Frida, and backend services; ensures consistency across environments
CI/CD	GitHub Actions	Automates testing and deployment; supports versioned delivery of updated security tools

Table 1.2. Layer-wise Tools and Technologies with Justification

Chapter 2

Literature Review

2.1 Overview

The continuous increase in Android malware incidents has led to extensive research into the domains of static and dynamic analysis, machine learning (ML), deep learning (DL), and hybrid approaches for vulnerability detection and prevention. In this section, we review the contributions and frameworks that align closely with the aims and objectives of SherlockDroid, an intelligent one-step solution designed to automate the process of reverse engineering, vulnerability analysis, and PoC generation for vulnerable or malicious Android APKs.

2.1.1 Machine Learning-Based Android Malware Detection:

Sharma and Kaul [9] carried an extensive review of Android malware detection mechanisms rooted in machine learning techniques. The authors highlight how Android malware, through various clever techniques such as code obfuscation and permission misuse, can trick and bypass traditional signature-based systems. The paper categorizes the detection methodologies into static, dynamic, and hybrid models and provides us with comparative evaluations among these. It emphasizes that analyzing behavioral patterns extracted from app metadata, permissions, and API usage, with the help of ML-based detection, has proved really effective. However, many challenges still exist, such as real-time processing, evolving malware signatures, and the lack of high-quality labeled datasets to help train ML models. This work aligns with SherlockDroid's use of LLMs and hybrid analysis for robust detection, while also motivating the inclusion of dynamic data flow tracking.

2.1.2 ML and DL Trends in Android and IoT Malware Detection:

Almobaideen et al. [10] presented a comparative review of recent ML and DL models that were applied for malware detection in Android and IoT environments. The study highlights the increasing complexity of malware and the limitations of traditional detection methodologies. It emphasizes the shift toward DL models (e.g., CNNs, RNNs) for feature extraction and classification in malware analysis. In addition, the authors reviewed various public datasets, highlighting their role in training robust classifiers. The review is valuable in identifying trends like dataset imbalance, overfitting in DL models, and generalization issues. SherlockDroid benefits from this by employing LLMs in place of traditional DL models for semantic analysis of app behaviors, using static and dynamic features.

2.1.3 Multimodal Static and Dynamic Detection Frameworks:

Faiz et al. [11] proposed a multimodal ML approach that combines static code analysis with dynamic behavior profiling for Android malware detection. Their architecture integrates static features like manifest attributes and API calls with dynamic traits such as system calls and runtime permissions. The paper demonstrates that multimodal architectures significantly outperform unimodal ones in terms of precision and recall. This dual-channel approach supports SherlockDroid's own architecture, which processes both disassembled code (via JADX[2]) and behavior logs for comprehensive vulnerability reporting and PoC generation.

2.1.4 Adversarial Attack Challenges in Android Malware Detection:

Ahmad et al. [12] introduced GEAAD, a framework for generating evasive adversarial attacks against Android malware detection systems. By leveraging opcode manipulation through GANs (Generative Adversarial Networks), their proposed model creates modified unique malware samples that evade existing classifiers. The study underscores the vulnerabilities in static detection systems and suggests integrating adversarial resilience into future frameworks. SherlockDroid can take advantage of the insights from this work by incorporating adversarial robustness techniques during the model fine-tuning phase or by identifying potentially evasive behaviors during static opcode analysis.

2.1.5 Systematic Reviews on Dynamic Analysis Techniques:

Sutter et al. [13] conducted a systematic literature review on dynamic analysis methods applied to Android applications. Their review outlines various dynamic analysis tools (e.g., instrumentation, taint analysis, syscall tracing) and discusses their applications in malware detection, privacy auditing, and behavioral profiling. The authors propose a taxonomy that classifies analysis objectives, tooling constraints, and coverage challenges. SherlockDroid's dynamic analysis component directly benefits from these findings by prioritizing methods like instrumentation-based syscall tracing and emulated environment setups for safe dynamic execution.

2.1.6 Static Analysis and ML Tool Trends:

Dahiya et al. [14] provided a survey on Android malware analysis, identifying static analysis as the most widely used technique in recent literature. Tools such as ApkTool, Androguard, and MobSF are frequently utilized for decompilation and permission extraction. The study also stresses the increasing application of ML and DL techniques, despite obstacles like code obfuscation, encrypted payloads, and dynamic code loading. SherlockDroid addresses these gaps by integrating deobfuscation steps and using LLMs to interpret hidden or dynamically loaded code segments more effectively than traditional ML classifiers.

2.1.7 Limitations in Existing Tooling and the Need for LLM Integration:

Tools like **FlowDroid** [15] and **QARK** [16] provide solid static analysis foundations but lack runtime behavior logging and dynamic response capabilities. **MobSF** [1] supports both static and limited dynamic analysis, generating detailed reports; however, its scalability problems and lack of generative AI support hinder deep interpretability in large-scale APK batches. **OpenAI Codex** and **VulnGPT** [17] are examples of LLMs contributing in security contexts, particularly in identifying insecure coding practices and generating PoCs. However, their models are generic and not fine-tuned for Android-specific pen testing, permissions, or PoC generation, limiting their utility in targeted malware analysis.

2.1.8 DroidSecureX

[18] combines static rule-based analysis with simulated syscall behaviors to detect malicious intent without actual app execution. Its approach is efficient in environments where dynamic analysis is not possible, but lacks adaptability to zero-day exploits and obfuscated logic, which SherlockDroid aims to handle using LLM reasoning and decompilation. **APK Analyzer** [19] focuses on manifest and permission analysis, offering quick and focused analysis; however, its limited features and lack of vulnerability insights or PoC attack vector generation make it insufficient for modern malware.

2.1.9 Bug Report Agent

[20] employs a Retrieval-Augmented Generation (RAG) model to explain general software bugs. While being valuable for desktop software QA, it lacks Android-specific context and fails to capture malicious behaviors or exploit chains embedded in APKs. **Cyber App** [21] detects and patches vulnerabilities in C/C++ using CWE and CVE rulesets, but does not support Java/Kotlin or APK-based application formats. These tools represent the gap SherlockDroid fills — combining Android-specific static/dynamic analysis with LLM-driven semantic understanding for accurate threat reporting and PoC automation.

2.1.10 Static and Dynamic Analysis in Android Penetration Testing:

According to Sir Arif Butt [22], Android application penetration testing fundamentally revolves around two key methodologies—*static analysis* and *dynamic analysis*.

Static analysis focuses on inspecting the application package (APK) without executing it. This process involves decompiling or disassembling the APK using tools such as **apktool**, **jadx**, and **dex2jar** to examine its internal components, including the `AndroidManifest.xml`, source code, and resource files. Through this approach, security researchers can identify vulnerabilities such as permission misuse, hardcoded credentials, and insecure API calls. Static analysis is considered safe for handling potentially malicious applications since it does not require app

execution. Sir Arif Butt [22] emphasizes that a thorough understanding of the Android software stack from the Linux kernel and native libraries to the application framework is essential for effectively interpreting static analysis results, particularly when analyzing Dalvik bytecode and manifest-level configurations.

In contrast, *dynamic analysis* involves executing the APK within a controlled or emulated environment to observe its real-time behavior. As described by Sir Arif Butt [22], environments such as the Android Virtual Device (AVD) emulator in Android Studio or tools like **MobSF** allow researchers to monitor runtime parameters such as system calls, inter-process communication, and network traffic. Dynamic analysis provides deeper insights into an application's operational behavior, including runtime permission usage, data leaks, and background service activity—vulnerabilities often missed during static examination. Additionally, instrumentation and Android Debug Bridge (ADB) utilities are used to trace function calls and analyze data flows dynamically.

Integrating both methodologies provides a more comprehensive security assessment. SherlockDroid builds upon this hybrid principle by combining static code inspection with dynamic behavioral profiling. This fusion enables detailed detection of malicious patterns, permission misuse, and anomalous runtime activities, aligning closely with best practices outlined by Sir Arif Butt [22].

2.1.11 Reverse Engineering Tools and Decompilation Advances

Modern reverse engineering tools such as **JADX**[2], **Dex2Jar**, and **Ghidra**[23] have become essential for analyzing Android applications by transforming DEX bytecode into human-readable Java code [24]. Despite these advancements, many tools struggle against modern obfuscation techniques, including control-flow flattening and packed binaries. **Ghidra**[23], an open-source suite maintained by the NSA, provides both DEX and native library analysis capabilities, yet it still demands substantial manual intervention for semantic reconstruction of code logic [25]. **Androguard** [26], a Python-based static analysis framework, supports disassembly, control-flow extraction, and APK signature validation but lacks automated interpretation of high-level behavior. **SherlockDroid** builds upon these foundations by integrating large language model (LLM)-driven semantic reasoning, allowing automatic interpretation of complex or obfuscated code paths that traditional decompilers cannot easily explain.

2.1.12 Use of LLMs and Generative AI in Cybersecurity

Recent works such as **LAMD** (Large Android Malware Detector) [27] have demonstrated the application of large language models in Android malware detection. LAMD employs contextual code understanding and chain-of-thought reasoning to identify malicious intent in complex APKs. Similarly, **TraceRAG** [28] introduces a retrieval-augmented generation (RAG) framework to produce human-readable malware behavior explanations by linking static features with dynamic traces. These studies reflect the growing synergy between cybersecurity and generative AI,

emphasizing interpretability and automation. However, most models remain limited to detection or explanation tasks without extending to exploit generation. **SherlockDroid** advances this integration by employing LLMs not only for semantic detection but also for automated proof-of-concept (PoC) generation and dynamic validation in Android-specific environments.

2.1.13 OWASP Top 10 for Mobile Applications

The **OWASP Mobile Top 10 (2024)** [29] identifies the most critical security risks for mobile applications, including **M1: Improper Credential Usage**, **M2: Inadequate Supply Chain Security**, and **M3: Insecure Authentication/Authorization**. These categories serve as a standardized benchmark for assessing and classifying mobile app vulnerabilities. Existing tools such as MobSF[1] and QARK[16] partially align with these categories but often lack contextual interpretation of interrelated flaws, such as how insecure data storage can be exploited through weak cryptography or unsafe communication channels. **SherlockDroid** addresses this by mapping vulnerability findings directly to OWASP categories and generating LLM-driven analytical summaries that explain the causal relationships between detected weaknesses, improving both interpretability and reporting accuracy.

2.1.14 Explainable AI (XAI) in Malware Detection

Explainable AI (XAI) has emerged as a critical area in security-focused machine learning, aiming to make automated detections more transparent and interpretable. **HuntGPT** [30] integrates LIME and SHAP methodologies to visualize feature contributions in anomaly detection systems, improving analyst trust in automated results. In Android malware detection, recent frameworks such as **TraceRAG** [28] adopt retrieval-augmented explanations to map code snippets to behavioral justifications. Despite this progress, most models lack contextual grounding across hybrid (static + dynamic) inputs. **SherlockDroid** incorporates XAI principles through natural language explanations generated by LLMs, clarifying why a sample is classified as malicious—e.g., highlighting dangerous permission patterns or encrypted payload execution—thereby bridging the gap between model output and human reasoning.

2.2 Gap Analysis

Lack of Exploit Generation:

Tools like MobSF[1] and QARK[16] stop at vulnerability detection and do not generate Proof-of-Concept (PoC) exploits. DroidSecureX[18] performs behavioral analysis but does not validate findings through exploitation.

SherlockDroid generates PoC exploits using LLMs for a subset of identified vulnerabilities and tests them in a controlled, emulated environment.

Absence of Natural Language Explanations:

Tools such as APK Analyzer[19] and FlowDroid[15] output technical data without offering understandable interpretations. Bug Report Agent[20] and Cyber App[21] show potential but are not tailored for Android apps.
SherlockDroid integrates LLMs to explain vulnerabilities, their consequences, and suggested fixes in simple, context-aware natural language.

Unstructured and Verbose Output Reports:
Many tools (e.g., MobSF[1], APK Analyzer[19]) produce long and unprioritized reports that are difficult for users to interpret, especially for beginners.
SherlockDroid filters analysis results and presents them in a ranked, human-readable format, emphasizing critical findings and suggested remediations.

Separation of Static and Dynamic Analysis:
Tools usually specialize in either static (MobSF[1], APK Analyzer[19]) or dynamic analysis (Frida[3], DroidSecureX[18]), rarely combining both in a unified pipeline.
SherlockDroid combines both static and dynamic analysis in a modular design to ensure comprehensive coverage of code and runtime behavior.

Lack of Interactive and Guided Learning:
Existing tools provide limited user interaction or learning support. While Cyber App[21] offers a chatbot interface, it is focused on C/C++ code and lacks Android context.
SherlockDroid offers an AI assistant that responds to user queries about APK structure, vulnerabilities, and remediations in real-time through a guided chatbot interface.

Limited Applicability to APK-Specific Structures:
General tools such as Cyber App[21] and Bug Report Agent[20] are not built to analyze Android APK files, missing important platform-specific issues.
SherlockDroid is purpose-built for APKs and focuses on common Android vulnerabilities, integrating both AI and mobile security standards (e.g., OWASP MASTG[8]).

Table 2.1. Comparison of SherlockDroid with Existing Android Security Analysis Tools

Feature	**MobSF** [1]	**QARK** [16]	**APK Analyzer** [19]	**FlowDroid** [15]	**DroidSecureX** [18]	**SherlockDroid** (Ours)
A. Analysis Capabilities						
A1. Static Analysis	✓	✓	✓	✓	✗	✓
A2. Dynamic Analysis	✓	✗	✗	✗	✓	✓
A3. Unified Analysis Pipeline	✗	✗	✗	✗	✗	✓
B. Exploit & Validation						
B1. PoC Exploit Generation	✗	✗	✗	✗	✗	✓
B2. Emulated Testing	✗	✗	✗	✗	✓	✓
C. User Experience & Interpretability						
C1. Natural Language Explanations	✗	✗	✗	✗	✗	✓
C2. Prioritized & Ranked Reports	✗	✗	✗	✗	✗	✓
C3. Interactive AI Assistant	✗	✗	✗	✗	✗	✓
C4. Real-time Query Support	✗	✗	✗	✗	✗	✓
D. Advanced Features						
D1. LLM Integration	✗	✗	✗	✗	✗	✓
D2. APK-Specific Focus	✓	✓	✓	✓	✓	✓
D3. OWASP MASTG Alignment	✓	✓	✗	✗	✓	✓

Chapter 3

User Stories & Epics

3.1 Epic: Platform Integration & Automation

Title: User Sign Up
Priority: High
User Story: As a new user, I want to create an account so that my credentials are securely stored and only authorized users can later log in.
Acceptance Criteria: • Given that a new user accesses the registration form, when valid details (username, email, password) are submitted, then the system should validate the inputs, hash the password using Django's password hashing (PBKDF2 with a unique salt) securely, store the user with their assigned role, and confirm success or redirect to login. • Given that the password is stored, it must never be saved in plain text and must only be verifiable through Django's authentication system. • Given invalid inputs (weak password, duplicate username/email), the system should reject the registration and display specific validation errors.

Table 3.1. User Sign Up

Title: User Login
Priority: High
User Story: As a registered user, I want to log in using my username/email and password so that authentication system verifies my identity securely and grants role-based access.
Acceptance Criteria: • Given that the user is registered, when valid credentials are entered, then the system should authenticate them using Django's authentication backend, verify the hashed password, and redirect to the appropriate dashboard based on their role. • Given that the credentials are invalid, the system should deny access and display a generic error message without revealing which field is incorrect. • Given multiple failed login attempts, the system should log them and optionally trigger a temporary account lockout.

Table 3.2. User Login

Title: Two-Factor Authentication
Priority: High
User Story: As a logged-in user, I want to complete a two-factor authentication step (via TOTP app or email OTP) so that even if my password is compromised, my account remains secure.
Acceptance Criteria: • Given that the user has entered correct login credentials, when 2FA is enabled, the system should prompt for a one-time code from an authenticator app or sent to the registered email. • Given that the code is valid and within its time limit, the system should grant access to the dashboard and features. • Given that the code is invalid or expired, the system should deny access, display an error, and log the failed attempt. • Given that 2FA is not enabled, the system should proceed with login after password verification but log the event for audit purposes.

Table 3.3. Two-Factor Authentication

Title: Subscription and Role-Based Access Integration
Priority: High
User Story: As a user, I want to subscribe to the APK analysis service using Stripe or JazzCash and have my access role automatically updated based on my subscription status, so that I can access premium platform features securely and seamlessly.
Acceptance Criteria: • Given that the user opts to activate a subscription, when payment is initiated, then the system should present options for Stripe and JazzCash and redirect the user to the selected payment gateway. • Given the payment process, when payment succeeds or fails, then the system should confirm the result, update the subscription status, and assign the appropriate role (e.g., free user, premium subscriber) accordingly. • Given an active subscription and assigned role, when the user accesses the dashboard or APK analysis features, then access should be granted or restricted based on the current role. • Given the subscription, when the user views their dashboard, then billing history and subscription expiry details should be clearly displayed.

Table 3.4. Subscription and Role-Based Access Integration

3.2 Epic: APK Analysis Engine

Title: APK Upload & Analysis via Dashboard
Priority: High
User Story: As an analyst, I want to automate APK upload and analysis via a web dashboard so that users can analyze apps efficiently.
Acceptance Criteria: • Given that a user has an APK file, When they drag and drop it onto the dashboard, Then the system should upload it and show progress. • Given that the APK is uploaded, When the user selects analysis type (static, dynamic, both), Then the appropriate analysis should start. • Given that the analysis is complete, When the user views the results, Then it should display timestamp, issues, severity, and action recommendations.

Table 3.5. APK Upload & Analysis

Title: Static Analysis of APKs
Priority: High
User Story: As a developer, I want to implement a secure static analysis engine so that APKs can be scanned for vulnerabilities, insecure configurations, and malicious code patterns without executing the application.
Acceptance Criteria: • Given that a user uploads an APK, when static analysis is triggered, then the system should securely extract and parse the manifest, DEX code, resources, and permission sets without writing extracted files to unprotected locations. • Given that the APK content is parsed, when the engine checks for vulnerabilities, then it should detect known issue types, including but not limited to: hardcoded API keys/secrets, insecure permissions, use of deprecated/unsafe APIs, exported components without protection, and presence of obfuscated or suspicious code blocks. • Given that code and resource analysis is performed, when signatures or patterns of known malware are found, then the system should flag them with a confidence score. • Given that vulnerabilities or risky configurations are identified, when the user views the report, then each finding should be tagged with severity (Critical/High-/Medium/Low), a brief description, technical details, and recommended remediation steps. • Given that no critical issues are found, when the report is generated, then the system should still provide a security posture summary and recommended best practices.

Table 3.6. Static Analysis of APKs

Title: Dynamic Analysis of APKs
Priority: High
User Story: As a developer, I want to implement a secure dynamic analysis system so that the runtime behavior of APKs can be observed in a sandboxed environment without risking the host system.
Acceptance Criteria: • Given an uploaded APK and a dedicated sandbox device, when the dynamic analysis tool is executed, then the APK should install and run in an isolated, non-persistent environment to prevent system compromise. • Given the app is running, when it performs actions, then the system should capture real-time logs, API calls, system events, and network traffic, ensuring all data is stored securely for later review. • Given that runtime data is collected, when the analysis process is complete, then the system should automatically compare behaviors against a predefined set of malicious behavior signatures and flag suspicious activities. • Given network analysis is performed, when outbound/inbound connections occur, then the system should log destination IPs, domains, protocols, and detect indicators of compromise (IoCs). • Given that the analysis session is finished, when the sandbox is reset, then all installed files, cached data, and configurations should be wiped to ensure a clean environment for the next analysis.

Table 3.7. Dynamic Analysis of APKs

Title: Correlation of Analysis Results
Priority: Medium
User Story: As a security researcher, I want to correlate static and dynamic results so that I can better detect potential malware patterns.
Acceptance Criteria: • Given that static and dynamic reports exist, When correlation module runs, Then it should merge and compare results. • Given merged data, When threats overlap or differ, Then the system should highlight overlaps and discrepancies. • Given all findings, When the summary is generated, Then it should include a confidence-based risk score.

Table 3.8. Correlation of Analysis

Static Analysis	**Dynamic Analysis**
Analyzes APK without execution (code inspection)	Analyzes APK behavior during runtime (executed in sandbox/emulator)
Input: APK file, source code, manifest, permissions	Input: APK installed on emulator or device
Process: Parse code, extract manifest, detect vulnerabilities like hardcoded keys, insecure permissions, known patterns	Process: Execute app in isolated environment, monitor API calls, system logs, network traffic
Detects potential vulnerabilities, coding flaws, and insecure configurations	Detects runtime behaviors such as suspicious network activity, unauthorized file access, or memory exploitation
Fast analysis, no need for runtime environment	Slower due to execution, requires emulator or real device setup
Output: Detailed static report with severity, location, and remediation suggestions	Output: Runtime logs, behavior report, flagged suspicious activities
Best for early vulnerability detection during development	Best for detecting runtime-specific issues and behavioral anomalies

Table 3.9. Flow-based Comparison of Static vs Dynamic Analysis

3.3 Epic: Exploitation and PoC Management

Title: Detect Malicious Apps on Networked Devices
Priority: High
User Story: As an analyst, I want to detect malicious apps on a vulnerable phone that connects to the same network so that I can identify compromised devices early.
Acceptance Criteria: • Given that a mobile device connects to the same LAN as the server, When the tool performs a network scan, Then it should identify all reachable devices by IP and MAC address. • Given that a device is discovered on the network, When the system has access (e.g., via ADB or agent), Then it should retrieve a list of installed applications from the device. • Given that the list of installed apps is available, When the system compares app hashes or package names against a database of known malicious APKs, Then it should flag any matches as threats. • Given that a malicious app is detected on a device, When the dashboard displays the analysis results, Then it should show the device details, threat level, and recommended remediation steps.

Table 3.10. Detect Malicious Apps

Title: Auto-generate Exploits from Vulnerabilities
Priority: Medium
User Story: As an analyst, after receiving vulnerability detection results from static and/or dynamic analysis, I want to provide the detailed report to a Large Language Model (LLM) and request automatic generation of attack vectors and exploits, so that I can accelerate exploitation testing.
Acceptance Criteria: • Given that vulnerabilities are identified in the analysis report, when the user issues a command to generate exploits, then the system should send the report to the LLM to produce corresponding attack vectors and template exploits with relevant metadata. • Given that exploit templates are generated, when the user views them, then each should include detailed code logic, environment setup instructions, and deployment steps for exploitation. • Given the generated exploits, when the user saves or modifies them, then the system must securely store them with version control and audit logging.

Table 3.11. Auto-generate PoC Exploits from Vulnerabilities

Title: Deliver PoC exploit to Device via ADB
Priority: Medium
User Story: As an operator, I want to deliver PoC exploit to a lab device using ADB so that I can begin exploitation.
Acceptance Criteria: • Given a PoC exploit file is ready, When ADB push is triggered, Then the PoC exploit should be transferred to the connected device. • Given the transfer is successful, When it completes, Then a confirmation message with device details and timestamp should be logged.

Table 3.12. Deliver PoC via ADB

Title: Execute exploit via Dashboard
Priority: Medium
User Story: As a developer, I want to trigger exploit execution via dashboard so that manual steps are minimized.
Acceptance Criteria: • Given exploits exist in the dashboard list, When a user clicks execute, Then the exploit should run and output should be streamed live. • Given that exploit executes, When it completes, Then device status should be updated and logs stored.

Table 3.13. Execute PoC

Title: Log Device Actions During Exploitation
Priority: Low
User Story: As a researcher, I want to log all device actions during exploitation so that I can generate detailed reports.
Acceptance Criteria: • Given an exploit session begins, When device actions happen (API/file/network), Then they should be logged with timestamps. • Given session logs are complete, When the user exports logs, Then it should allow CSV or JSON format with session ID.

Table 3.14. Log Device Actions

3.4 Epic: Wi-Fi Network Penetration Testing & Local Network Exploitation

Title: Wi-Fi Network Vulnerability Assessment
Priority: High
User Story: As a penetration tester, I want to analyze vulnerabilities in Wi-Fi networks so that I can capture sensitive data and crack weak passwords.
Acceptance Criteria: • Given a target Wi-Fi network using WPA2-PSK, when I capture the handshake using `airodump-ng`, then I should extract the PMKID or handshake packets for offline cracking. • Given a weak password exists in my wordlist, when I use `aircrack-ng` or `hashcat`, then I should successfully retrieve the WPA2 key. • Given an active Wi-Fi network, when I launch a deauthentication attack with `aireplay-ng --deauth`, then clients should be disconnected and a handshake recaptured.

Table 3.15. Wi-Fi Network Vulnerability Assessment

Title: Wi-Fi Network Exploitation
Priority: Low
User Story: As a penetration tester, I want to exploit vulnerable Wi-Fi networks so that I can demonstrate attack scenarios against connected devices.
Acceptance Criteria: • Given a rogue AP is set up with `airbase-ng`, when a victim connects, then I should capture credentials or perform a Man-in-the-Middle interception.

Table 3.16. Wi-Fi Network Exploitation

Title: Exploiting Devices on the Same Wi-Fi LAN
Priority: Medium
User Story: As a security researcher, I want to identify and exploit vulnerabilities in mobile or IoT devices connected to the same Wi-Fi LAN so that I can simulate real-world post-connection attack scenarios.
Acceptance Criteria: • Given a device is on the same LAN, when I perform an ARP spoofing attack using `ettercap` or `BetterCAP`, then I should intercept its network traffic. • Given an intercepted HTTP session, when I use `Wireshark` or `sslstrip`, then I should retrieve session cookies or credentials. • Given an open network share or exposed service, when I scan using `nmap`, then I should identify vulnerable ports and services for exploitation. • Given a captured session, when I replay authentication tokens, then I should be able to hijack the user's active session.

Table 3.17. Exploiting Devices on the Same Wi-Fi LAN

3.5 Epic: Advanced Threat Simulation

Title: Simulate Backdoor Persistence
Priority: Medium
User Story: As an attacker, I want to simulate a backdoor that maintains persistence by hooking into system services or events, so that I can test how the backdoor survives device reboots and user sessions in the target environment.
Acceptance Criteria: • Given that a backdoor payload is deployed on the device or emulator, when the system reboots, then the payload should auto-execute without requiring user interaction. • Given that hooks or triggers are configured (e.g., boot completion, screen unlock), when those events occur, then the backdoor should activate and establish control or communicate back. • Given the persistence simulation runs, when system or application logs are reviewed, then all activation attempts and outcomes (success/failure) should be properly logged and accessible for analysis.

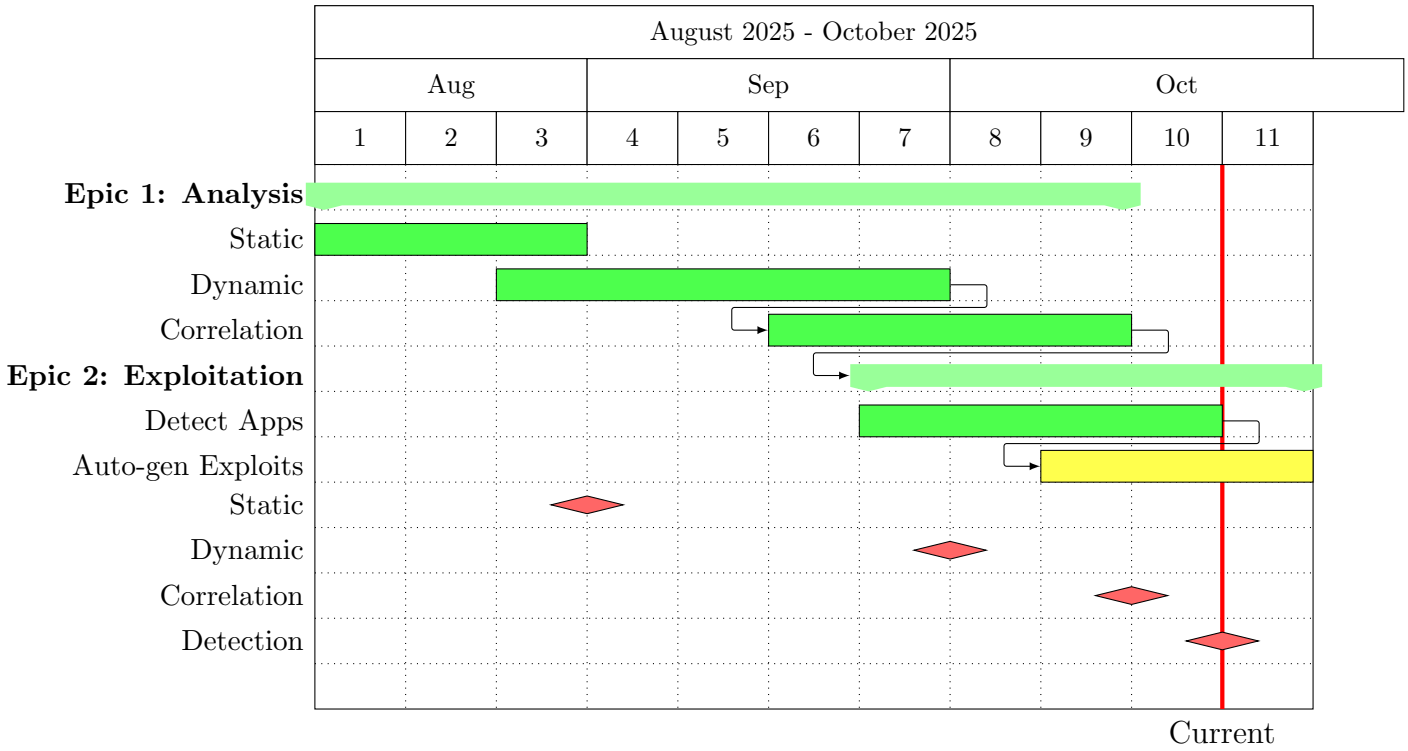
Table 3.18. Simulate Backdoor Persistence

Title: C2-Based System Surveillance
Priority: Medium
User Story: As an attacker, I want to maintain C2 access so I can capture credentials and access the camera or microphone for surveillance.
Acceptance Criteria: • Given a C2 session is active, when credential harvesting is triggered, then stored passwords or tokens should be retrieved. • Given access to hardware, when commands are issued, then mic or camera feeds should be activated. • Given system usage is being monitored, when the user interacts with the device, then activity logs should be recorded.

Table 3.19. Simulate Command & Control (C2) Channels

3.6 SPRINTS - Project Timeline

3.6.1 Phase 1: Core Analysis & Exploitation (August - October 2025)

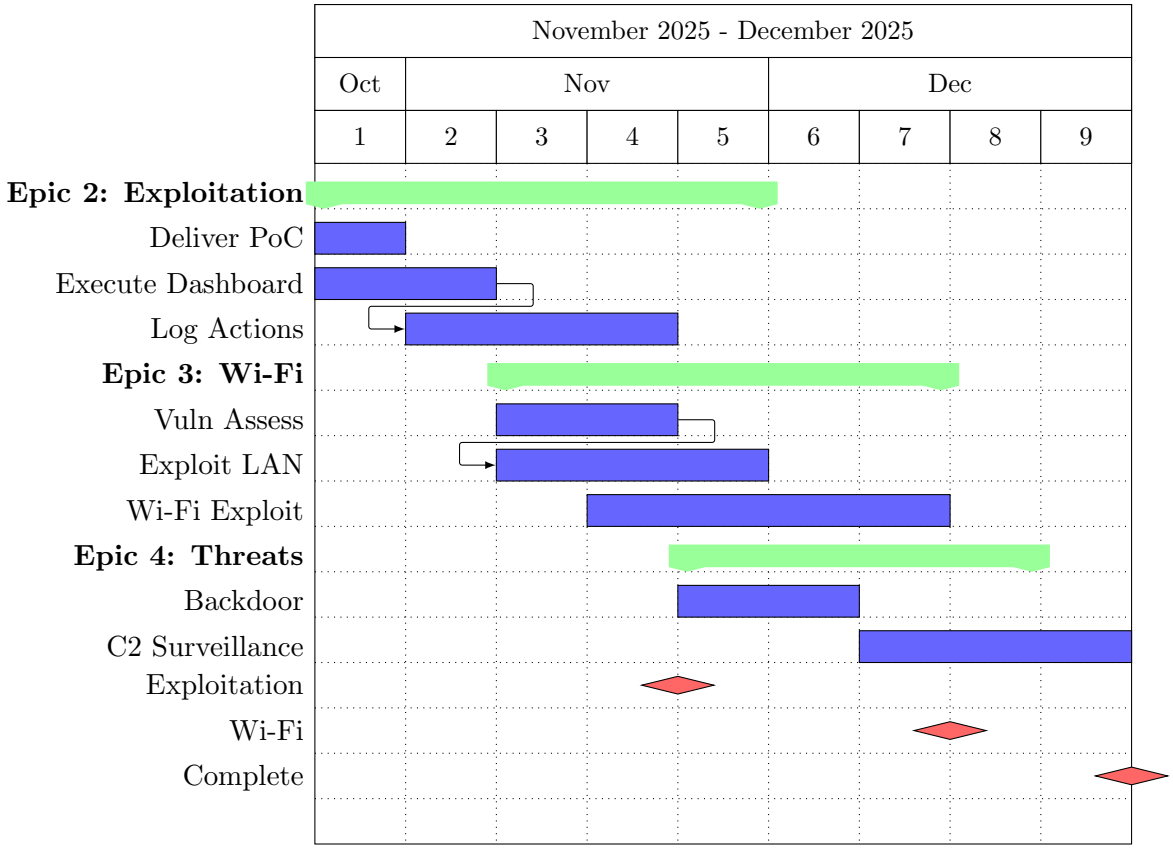


Status Summary (Phase 1):

- **Completed:** Static Analysis, Dynamic Analysis, Correlation of both static and dynamic analysis, Detect Apps on mobile devices that appear in the same LAN.
- **In Progress:** Auto-generate Exploits

- **End:** October 2025

3.6.2 Phase 2: Exploitation & Advanced Testing (November 2025 - December 2025)



Status Summary (Phase 2):

- **Start Date:** November 2025
- **Expected Completion:** End of December 2025
- **Total Duration:** 2 months (8 weeks)
- **Dependencies:** Requires completion of Phase 1 detection capabilities

Overall Project Timeline:

- **Project Start:** August 2025
- **Project End:** December 2025
- **Total Duration:** 5-6 months

Chapter 4

Blue Team Operations: Android Application Security Testing

4.1 Introduction

This chapter presents a comprehensive blue team approach to Android application security testing, focusing on three core phases: network discovery and APK acquisition, static code analysis, and dynamic runtime analysis. The methodology combines automated scanning tools with Large Language Model (LLM)-powered analysis to identify security vulnerabilities at scale. This work demonstrates a complete security assessment pipeline from initial reconnaissance through detailed vulnerability reporting.

4.2 Network Discovery and APK Acquisition

4.2.1 Overview

The first phase of the security assessment involves discovering Android devices on the local network and extracting application packages (APKs) for analysis. This automated reconnaissance approach enables blue teams to inventory Android devices and their installed applications efficiently.

4.2.2 Architecture and Implementation

The Android Device Scanner is implemented as a Python-based tool optimized for Windows environments, utilizing the Android Debug Bridge (ADB) protocol for device communication. The scanner operates in three distinct modes:

- **Mode 1: Network Discovery** - Identifies ADB-enabled devices on the local network
- **Mode 2: Device Enumeration** - Scans specific devices for installed applications
- **Mode 3: APK Extraction** - Retrieves application packages for offline analysis

Network Discovery Mechanism

The scanner employs a multi-layered approach to device discovery:

1. **ARP Cache Analysis**: Parses the Windows ARP table to identify active hosts within the target subnet

2. **Port Scanning**: Probes TCP port 5555 (ADB's default network port) to identify debug-enabled devices

3. **Fallback Ping Sweep**: Implements concurrent ICMP probes using ThreadPoolExecutor for rapid subnet scanning

Device Fingerprinting

Upon discovering an ADB-enabled device, the scanner executes the following device property queries:

```
ro.product.manufacturer      Device manufacturer
ro.product.model             Device model identifier
ro.build.version.release     Android OS version
```

4.2.3 Technical Implementation Details

The scanner leverages Windows-native commands including arp -a for MAC address resolution and PowerShell's Test-NetConnection for port scanning, ensuring compatibility with enterprise Windows environments. For APK extraction, the tool handles both standard applications and split APK packages, automatically creating organized directory structures. The stateful scanning approach maintains connection context through APKAnalysisState tracking, while priority package detection specifically targets known vulnerable applications like DIVA and InsecureBank for focused security assessment.

This automated discovery phase provides blue teams with critical visibility into potentially vulnerable Android devices that have ADB debugging enabled—a common security misconfiguration. By inventorying installed applications and extracting APKs for static analysis, security teams can identify shadow IT applications, detect known vulnerable packages, and build comprehensive application risk profiles. The tool's ability to operate at scale makes it particularly valuable for enterprise environments where manual device enumeration would be impractical.

4.2.4 Operational Workflow

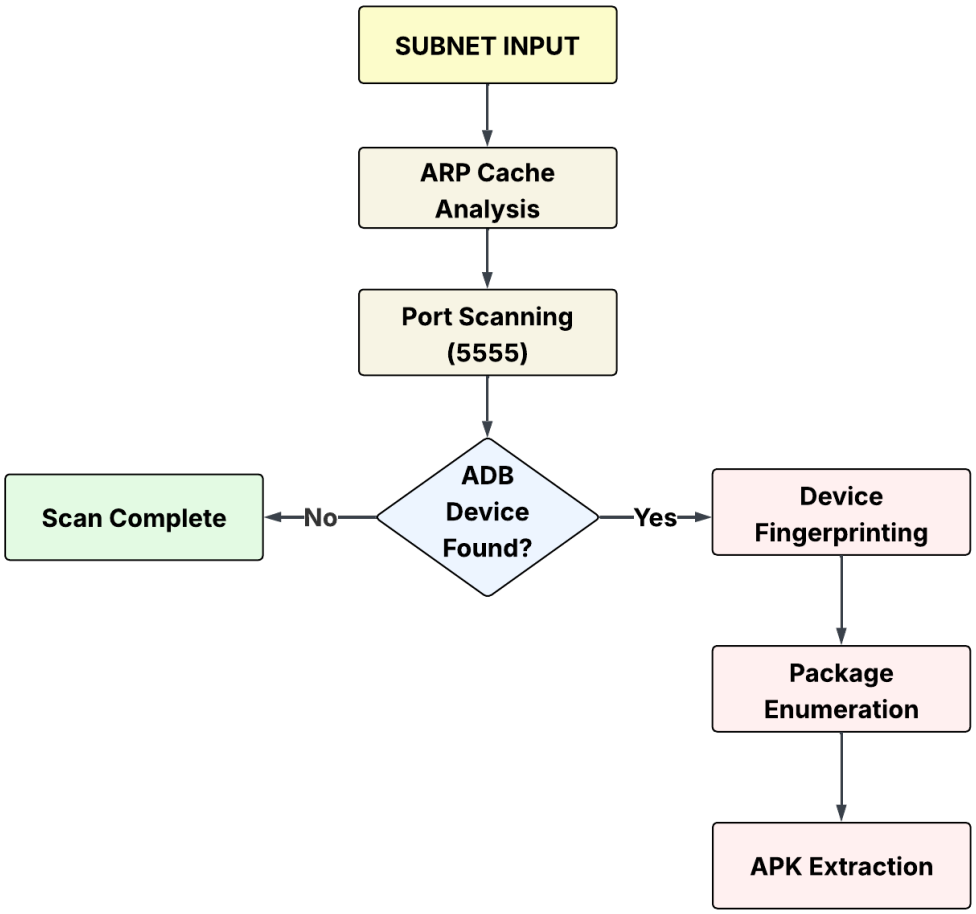


Fig. 4.1. Automated network discovery workflow depicting the sequential stages of scanning, enumeration, and asset classification for security analysis.

4.2.5 Security Considerations

The scanner implements several security measures:

- **No Window Creation**: Uses `CREATE_NO_WINDOW` flag to prevent terminal spawning
- **Timeout Protection**: Prevents hanging on unresponsive devices
- **Error Isolation**: Failed extractions don't terminate the entire scan
- **Credential Management**: No credentials stored; relies on pre-authorized ADB connections

4.3 Static Analysis with LLM-Powered Code Review

4.3.1 Overview

The static analysis phase employs a three-stage pipeline: APK structure scanning, LLM-powered file analysis, and results aggregation. This approach leverages GPT-4o Mini for intelligent security vulnerability detection across diverse file types within the APK structure.

4.3.2 APK Structure Analysis

File Classification System

The scanner implements context-aware file classification, where folder hierarchy takes precedence over file extensions:

- **Priority 1 - Special Files**: AndroidManifest.xml, classes.dex, resources.arsc
- **Priority 2 - Folder Context**: Files within `smali/`, `lib/`, `assets/` directories
- **Priority 3 - Extension Mapping**: File type based on extension (`.so`, `.dex`, `.xml`, etc.)

Analysis Categories

The system classifies files into 15 distinct analysis types:

Table 4.1. File Analysis Categories and Security Relevance

Category	Description	Security Relevance
`smali_code`	Decompiled bytecode	High - Contains application logic and potential vulnerabilities
`android_manifest`	App permissions & components	Critical - Defines security boundaries
`native_libs`	Compiled .so libraries	High - Binary analysis for exploits
`config_files`	YAML/JSON/Properties	Medium - Configuration vulnerabilities
`dex_bytecode`	Dalvik executable	High - Core application code
`certificates`	Signing certificates	Critical - Authentication integrity
`database_files`	SQLite databases	High - Data exposure risks
`web_files`	HTML/JS/CSS	Medium - XSS and injection vectors

4.3.3 LLM Integration Architecture

Multi-API Load Balancing

The analyzer implements dual-API architecture for token optimization:

- **API Key 1** (18,000 token limit):
  - Handles files under 20,000 tokens

  - Rate-limited to 10 concurrent requests
  - Priority for smaller configuration files
- **API Key 2** (50,000 token limit):
  - Processes large files (20k+ tokens)
  - Rate-limited to 5 concurrent requests
  - Handles complex smali and native code

Prompt Engineering

Each file type receives specialized prompts optimized for security analysis:

- **Smali Code Analysis Prompt** focuses on:
  - Hardcoded secrets (API keys, passwords, tokens)
  - Insecure network communications
  - SQL injection vectors
  - Intent-based security issues
- **Android Manifest Analysis Prompt** examines:
  - Dangerous permission requests
  - Exported component vulnerabilities
  - Debuggable flags in production
  - Network security configuration

4.3.4 LangGraph Parallel Processing

Dynamic Graph Construction

The system builds a custom LangGraph workflow for each APK:

```
START     initialize     [file_node_1, file_node_2, ..., file_node_N
   ]     aggregate     END
```

State Management

The `APKAnalysisState` TypedDict maintains:

```
{
    "apk_structure": APKStructure,           # Immutable scan
   results
    "file_analysis_results": Dict,           # Accumulated findings
   (merge function)
```

```
    "files_processed": int,                    # Counter (add function)
    "failed_files": List[str],                 # Failed file paths (extend function)
    "final_security_report": Dict,             # Aggregated report
    "analysis_start_time": str,                # ISO timestamp
    "total_files": int                         # Total file count
}
```

The LangGraph framework enables dynamic parallel processing by constructing customized workflows for each APK. The graph begins with initialization, spawns parallel analysis nodes for each file within the APK, and concludes with aggregation of results. This architecture allows concurrent processing of multiple files while maintaining shared state through the APKAnalysisState dictionary, which tracks progress, accumulates findings, and handles failures. The state management system uses specialized functions for different data types—merge for dictionaries, add for counters, and extend for lists—ensuring thread-safe operations during parallel execution. This approach significantly accelerates static analysis by processing files concurrently rather than sequentially, while maintaining comprehensive audit trails and error handling.

The combination of dynamic graph construction and robust state management provides a scalable foundation for analyzing complex Android applications with varying structures and file compositions.

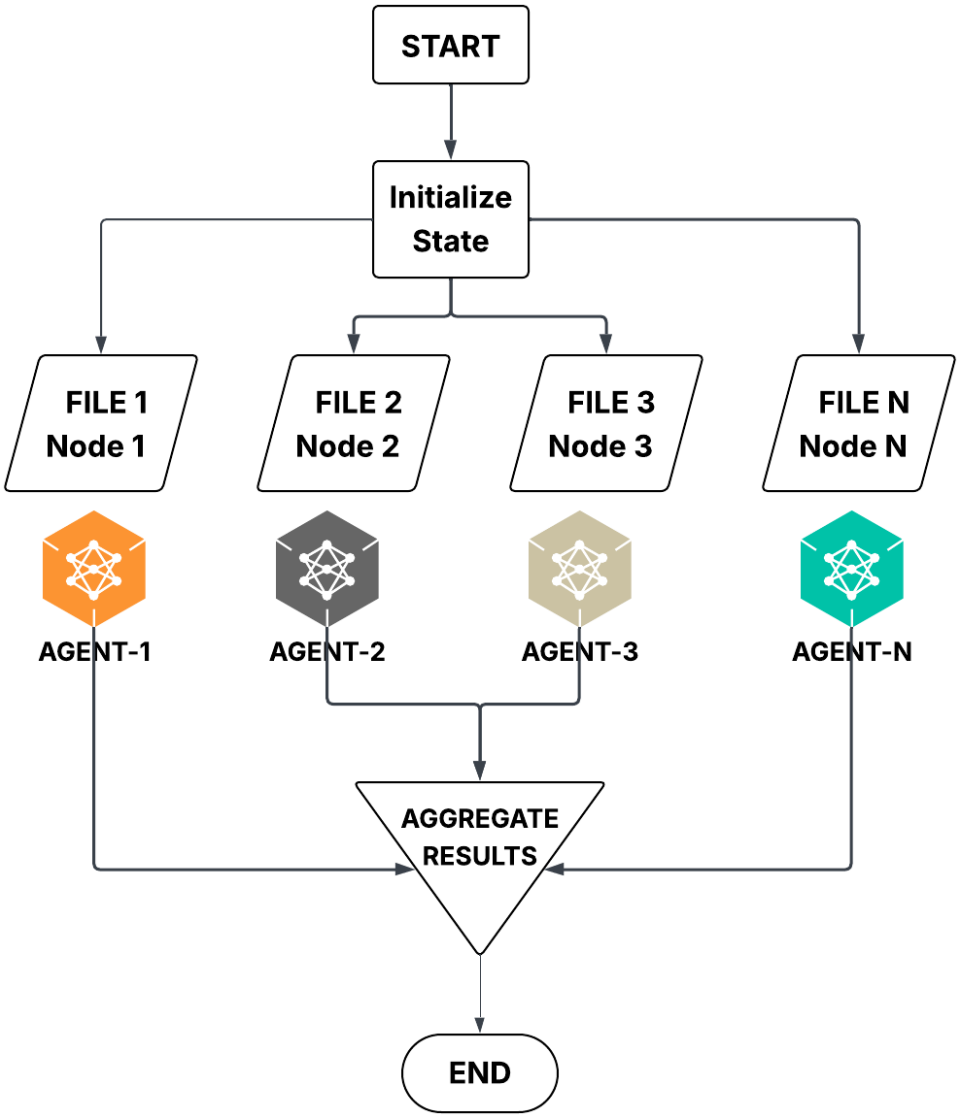


Fig. 4.2. Dynamic LangGraph workflow construction that processes each APK into file nodes for customized static analysis.

4.4 Dynamic Analysis with Frida Instrumentation

4.4.1 Overview

Dynamic analysis complements static analysis by observing application behavior at runtime. This phase employs Frida—a dynamic instrumentation framework—to hook critical Android APIs and capture security-relevant events including cryptographic operations, network communications, data storage, and reflection usage.

4.4.2 Instrumentation Architecture

Hook Categories

The dynamic analysis implements five specialized hook modules:

- `crypto_hooks.js` - Cryptographic API monitoring
- `network_hooks_fixed.js` - Network communication tracking
- `diva_hook.js` - Application lifecycle and data storage
- `reflection_hooks.js` - Dynamic code loading and execution
- `safe_credentials_hook.js` - User input and credential capture

4.4.3 Cryptographic Operations Monitoring

Cipher API Hooks

The crypto hooks target `javax.crypto.Cipher` to capture:

- **Cipher.getInstance(algorithm)**:
  - Captures encryption algorithm selection
  - Identifies weak algorithms (DES, RC4)
  - Logs: {`event: "Cipher.getInstance", alg: "AES/CBC/PKCS5Padding"`}
- **Cipher.init(opmode, key)**:
  - Monitors cipher initialization
  - Captures operation mode (ENCRYPT_MODE=1, DECRYPT_MODE=2)

4.4.4 Network Communication Analysis

Multi-Library Support

The network hooks implement adaptive detection for multiple HTTP libraries:

```
1 // HttpURLConnection Hooks
2 setRequestMethod(method)                    Captures HTTP verb
3 setRequestProperty(key, value)              Logs HTTP headers
4 connect()                                   Records connection
     establishment
5
6 // OkHttp3 Hooks
7 Request.Builder.build()                     Captures URL, method,
     headers
8
```

```
// Socket-Level Fallback
Socket.<init>(host, port)                    Logs raw socket
    connections
```

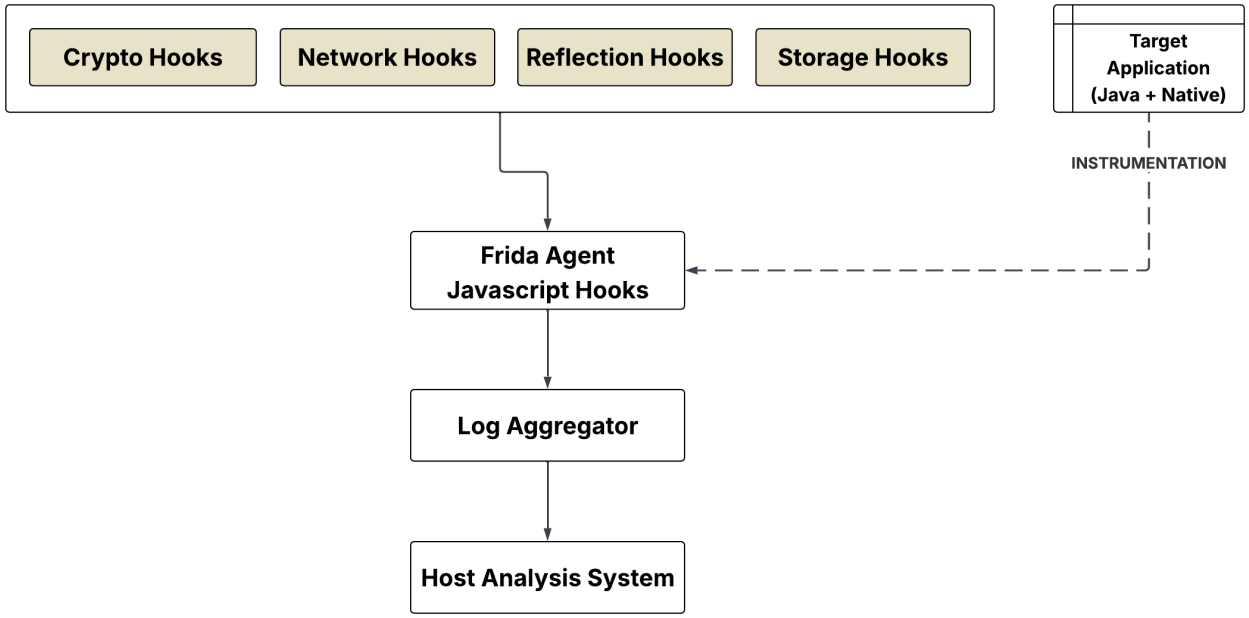


Fig. 4.3. Dynamic Security Instrumentation Architecture with runtime Application Monitoring via Frida Instrumentation

4.5 Report Generation and Vulnerability Scoring

4.5.1 Overview

The final phase synthesizes static and dynamic analysis results into executive-ready security reports. This involves LLM-powered report generation, professional PDF formatting, and interactive visualizations.

4.5.2 Report Generation Pipeline

JSON Data Structuring

The aggregated analysis results conform to a standardized schema:

```
{
    "analysis_metadata": {
        "analysis_start_time": "ISO-8601 timestamp",
        "analysis_end_time": "ISO-8601 timestamp",
        "apk_path": "file path",
        "overall_risk_assessment": "CRITICAL|HIGH|MEDIUM|LOW"
    },
```

```
    "executive_summary": {
        "total_findings": integer,
        "critical_findings": integer,
        "high_findings": integer,
        "medium_findings": integer,
        "low_findings": integer
    }
}
```

4.5.3 Vulnerability Scoring Methodology

CVSS v3.1 Integration

Each finding receives a CVSS v3.1 score based on:

- **Base Metrics**:
  - Attack Vector (AV): Network/Adjacent/Local/Physical
  - Attack Complexity (AC): Low/High
  - Privileges Required (PR): None/Low/High
  - User Interaction (UI): None/Required

Risk Level Categorization

Findings are categorized using severity thresholds:

Table 4.2. Vulnerability Severity Classification and Response Timelines

Score Range	Severity	Response Timeline
9.0-10.0	CRITICAL	24-48 hours
7.0-8.9	HIGH	1-2 weeks
4.0-6.9	MEDIUM	1 month
1.0-3.9	LOW	3 months

4.5.4 PDF Report Generation

Professional Styling

The ReportLab-based PDF generator implements custom paragraph styles:

- **TitleStyle**: 28pt Helvetica Bold, white on dark background
- **CriticalStyle**: Bold text on red background
- **HighStyle**: Bold text on orange background
- **MediumStyle**: Bold text on yellow background

- **LowStyle**: Bold text on green background

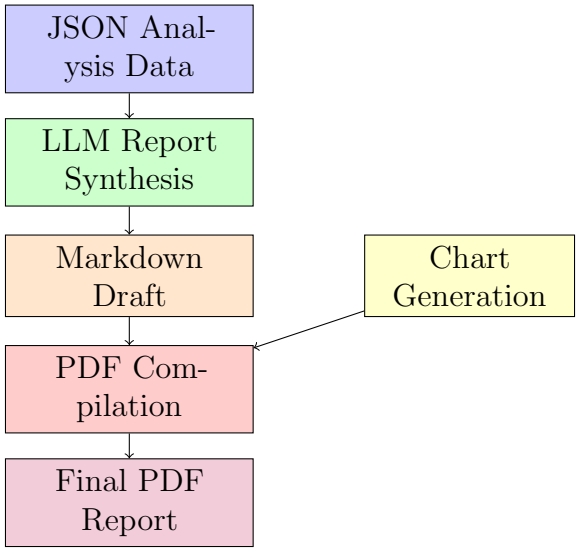


Fig. 4.4. Report Generation Pipeline

4.6 Integration and Operational Deployment

4.6.1 End-to-End Workflow

The complete security assessment workflow integrates all phases:

1. **Reconnaissance**: Network discovery identifies target devices
2. **Acquisition**: APKs extracted from discovered devices
3. **Static Analysis**: LLM-powered code review identifies potential vulnerabilities
4. **Dynamic Analysis**: Frida hooks capture runtime behavior
5. **Correlation**: Static and dynamic findings are cross-referenced
6. **Reporting**: Comprehensive security report generated

4.6.2 Continuous Integration

The toolchain can be integrated into CI/CD pipelines:

```
# Automated APK analysis in CI
python android_scanner.py --scan $DEVICE_IP --output results.json
python llm_analyzer.py --input results.json --output report.json
python report_generator.py --input report.json --output final_report
    .pdf
```

4.7 Conclusion

This chapter presented a comprehensive blue team methodology for Android application security testing, combining automated network reconnaissance, LLM-powered static analysis, and Frida-based dynamic instrumentation. The approach demonstrates how modern AI technologies can be leveraged to scale security assessments while maintaining analytical depth.

Key contributions include:

1. **Automated Discovery Pipeline**: Efficient network-based APK acquisition
2. **LLM-Powered Analysis**: Intelligent code review at scale using GPT-4o Mini
3. **Parallel Processing Architecture**: LangGraph-based concurrent file analysis
4. **Comprehensive Dynamic Instrumentation**: Multi-layer Frida hooks covering crypto, network, storage, and reflection
5. **Professional Reporting**: Executive-ready PDF reports with strategic recommendations
6. **Zero-Finding Innovation**: Positive framing of security achievements

References

[1] MobSF, *Mobile security framework (mobsf)*, https://github.com/MobSF/Mobile-Security-Framework-MobSF, Accessed: 2025-06-29, 2024 (cited on pp. 3, 9, 13, 14, 19, 21–23).

[2] JADX Developers, *Jadx – dex to java decompiler*, https://github.com/skylot/jadx, Accessed: 2025-06-29, 2025 (cited on pp. 3, 9, 13, 14, 18, 20).

[3] Frida, *Frida: Dynamic instrumentation toolkit*, https://frida.re/, Accessed: 2025-06-29, 2024 (cited on pp. 3, 13, 22).

[4] Payatu, *DIVA - Damn Insecure and Vulnerable App for Android*, https://github.com/payatu/diva-android, Accessed: 2025-06-29, 2020 (cited on p. 3).

[5] D. Shetty, *Android insecurebankv2*, https://github.com/dineshshetty/Android-InsecureBankv2, Accessed: 2025-06-29, 2018 (cited on p. 3).

[6] O. P. Contributors, *Androgoat - android app to demonstrate common vulnerabilities*, https://github.com/ashishb/android-goat, Accessed: 2025-06-30, 2016 (cited on p. 3).

[7] G. Community, *Damn vulnerable bank*, https://github.com/prashant-sg/Damn-Vulnerable-Bank, Accessed: 2025-06-30, 2020 (cited on p. 3).

[8] O. Foundation, *Owasp mobile app security test suite (mastg)*, https://github.com/OWASP/owasp-mastg, Accessed: 2025-06-30, 2023 (cited on pp. 3, 22).

[9] M. Sharma and A. Kaul, "A review of detecting malware in android devices based on machine learning techniques," *Security and Privacy*, 2023, Accessed: 2025-07-06 (cited on p. 17).

[10] W. Almobaideen *et al.*, "Comprehensive review on machine learning and deep learning techniques for malware detection in android and iot devices," *Journal of Ambient Intelligence and Humanized Computing*, 2025, Accessed: 2025-07-06 (cited on p. 17).

[11] A. Faiz *et al.*, "A multimodal machine learning approach for android malware detection: Static code analysis," *Pakistan Journal of Computer Science*, 2024, Accessed: 2025-07-06 (cited on p. 18).

[12] N. Ahmad *et al.*, "Geaad: Generating evasive adversarial attacks against android malware defense," *Journal of Cybersecurity*, 2025, Accessed: 2025-07-06 (cited on p. 18).

[13] T. Sutter *et al.*, "Dynamic security analysis on android: A systematic literature review," *IEEE Access*, 2025, Accessed: 2025-07-06 (cited on p. 18).

[14] A. Dahiya, S. Singh, and G. Shrivastava, "Android malware analysis and detection: A systematic review," *Security and Communication Networks*, 2024, Accessed: 2025-07-06 (cited on p. 18).

[15] FlowDroid Developers, *FlowDroid: Taint Analysis for Android Applications*, `https://github.com/secure-software-engineering/FlowDroid`, Accessed: 2025-06-29, 2023 (cited on pp. 19, 22, 23).

[16] LinkedIn Security, *Qark - quick android review kit*, `https://github.com/linkedin/qark`, Accessed: 2025-06-29, 2020 (cited on pp. 19, 21, 23).

[17] OpenAI, *Gpt-4 technical report*, `https://openai.com/research/gpt-4`, Accessed: 2025-06-29, 2024 (cited on p. 19).

[18] DroidXadmin, *Droidsecurex v2 – ai-powered android apk analyzer using simulated syscall data and static rules*, `https://huggingface.co/spaces/DroidXadmin/DroidSecureX_v2`, Accessed: 2025-06-29, 2024 (cited on pp. 19, 21–23).

[19] abubasith86, *Apk analyzer – upload an apk file for security analysis*, `https://huggingface.co/spaces/abubasith86/apk_analyzer`, Accessed: 2025-06-29, 2024 (cited on pp. 19, 22, 23).

[20] prakshhari197, *Bug report analysis agent using rag*, `https://huggingface.co/spaces/prakshhari197/bug_repots_analysic_agent`, Accessed: 2025-06-29, 2024 (cited on pp. 19, 22).

[21] PeepDaSlan9, *Cyber app – ai-powered code vulnerability identification and fixing using cwe/cve standards*, `https://huggingface.co/spaces/`

PeepDaSlan9/B2BMGMT_cyber_app, Accessed: 2025-06-29, 2024 (cited on pp. 19, 22).

[22] A. Butt, *Android app pen-testing i (handout 2.12)*, Department of Computer Science, Forman Christian College University, Lecture Handout, 2025 (cited on pp. 19, 20).

[23] K. Wright and J. Patel, "Advances in reverse engineering: Evaluating modern apk decompilers," *Journal of Mobile Security Research*, vol. 14, no. 2, pp. 45–57, 2023 (cited on p. 20).

[24] KDAB, *Reverse engineering android apps*, KDAB website, Discusses usage of JADX, Dex2Jar, native decompilation, obfuscation issues, 2023 (cited on p. 20).

[25] D. /. ASRP, *Static analysis — android security research playbook*, Online resource, 2023 (cited on p. 20).

[26] D. Bernard and L. Rousseau, "Androguard: A comprehensive framework for android reverse engineering," in *Proceedings of the International Conference on Digital Forensics*, ACM, 2022, pp. 88–95 (cited on p. 20).

[27] X. Qian *et al.*, "Lamd: Context-driven android malware detection and classification with llms," *arXiv preprint*, 2025 (cited on p. 20).

[28] G. Zhang *et al.*, "Tracerag: A llm-based framework for explainable android malware detection and behavior analysis," *arXiv preprint*, 2025 (cited on pp. 20, 21).

[29] O. Foundation, *Owasp mobile top 10 — the ten most critical mobile security risks*, https://owasp.org/www-project-mobile-top-10/, Accessed: 2025-10-07, 2023 (cited on p. 21).

[30] T. Ali and P. Kostakos, "Huntgpt: Integrating machine learning-based anomaly detection and explainable ai with large language models," *arXiv preprint*, 2023 (cited on p. 21).