NFSU

National Forensic Sciences University

Knowledge | Wisdom | Fulfilment

An Institution of National Importance
(Ministry of Home Affairs, Government of India)

PROJECT REPORT

ON

"WHATSZAP"

Submitted To

Department of Cyber Security & Digital Forensics

National Forensic Sciences University

For partial fulfilment for the award of degree

MASTER OF TECNOLOGY

In

COMPUTER SCIENCE ENGINEERING CYBER SECURITY

Submitted By

Deepanshu Sharma

102CTBMCS2122036

Under the Supervision of

DR. ARCHANA PATEL

SCHOOL OF CYBER SECURITY & DIGITAL FORENSIC

National Forensic Sciences University,

Delhi Campus, New Delhi – 110085, India

Dec 2025

DECLARATION

I hereby declare that the thesis entitled “WHATSZAP” is a research work done by me and no part of the thesis has been presented earlier and will be presented for any degree, diploma or similar title at any other institute/university.

DEEPANSHU SHARMA
102CTBMCS21220036
BTech MTech CS
2021 - 2026

Date:
Place: Delhi

ORIGNALITY REPORT CERTIFICATE

I certify that

a. The work contained in the dissertation is original and has been done by myself under the supervision of my supervisor.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

d. Whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the dissertation and giving their details in the references.

e. Whenever I have quoted written materials from other sources and due credit is given to the sources by citing them.

f. From the plagiarism test, it is found that the similarity index of whole dissertation within 10% and single paper is less than 10 % as per the university guidelines.

Deepanshu Sharma
102CTBMCS2122036

Date:
Place: Delhi

Forwarded by

Dr. Archana Patel

CERTIFICATE

This is to certify that the work contained in the dissertation entitled **"WhatsZap"**, submitted by **Deepanshu Sharma (Enroll. No.: 102CTBMCS2122036)** for the award of the degree of **Bachelors & Master of Technology in Computer Science Engineering CyberSecurity** to the **National Forensic Sciences University ,**is a record of bonafide research works carried out by him under my supervision and guidance

Dr. Archana Patel
Assistant Professor
Department Of CyberSecurity & Digital Forensic
National Forensic Sciences University Delhi Campus
Delhi, India

Date:
Place: Delhi

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my mentor, Dr. Archana Patel, for her invaluable guidance, constant encouragement, and insightful feedback throughout the development of Whatszap. Her support at every stage of this project not only helped me overcome challenges but also enhanced my understanding and confidence.

I am also deeply thankful to my friends and juniors who contributed in various ways—be it through brainstorming ideas, testing features, identifying bugs, or simply motivating me to keep going. Their help, time, and cooperation played a crucial role in shaping this project into its final form. Without their collective support, the successful completion of Whatszap would not have been possible.

With Sincere Regards,

DEEPANSHU SHARMA
102CTBMCS21220036
BTech MTech CS
2021 - 2026

ABSTRACT

Social media sites have emerged as an important dissemination channel for the integration of social engineering and malicious code distribution; in the Indian context, malicious APKs are distributed via trusted formats such as wedding cards. The proposed work describes an Android security framework for protecting against APK-based attacks spread via social media sites, generic in terms of design but implemented and assessed for the popular WhatsApp service. It always keeps track of download links from social media sites and the corresponding message context. Once an APK is downloaded, there is an immediate start of an "overlay" for 10 seconds, temporarily blocking the accidental install process while scanning the downloadable binary for potential malicious intent in an "on-device" scanner incorporating both "static" and "dynamic" approaches to check for known-signature APK attacks combined with heuristic analysis and "contextual" message classification informed by natural language features to segregate benign from malicious invitation messages with an emphasis on local analysis with "explicit" user consent and minimal "accessibility" and "storage" permissions to curtail privacy vulnerabilities with an assessment of detection accuracy, false positive rate, and scan time overhead with an evaluation of "usability" lag from the overlay feature for a comprehensive set of curated APK data from social media sites and benign media sharing scenarios.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Introduction and Problem Summary

WhatsApp is a primary channel for personal and group communication in India to share messages, photos, documents, and event invites. The attackers will leverage this high trust by sending malicious Android Package (APK) files camouflaged as benign content-most notably social-engineering lures such as wedding invitations, event flyers, or utility updates. Since Android allows side loading apps outside the Play Store, when a user taps on an APK received over WhatsApp, the action inadvertently leads to the installation of malware that steals data, escalates privileges, or persists on the device. The core problem this project addresses is: the pre-installation delivery of malware via WhatsApp, whereby adversaries deliver payloads as files attached to messages that are contextually plausible. Traditional mobile defences either rely on signature databases that lag new malware or detect threats well after installation, which is too late for effective prevention. Also, existing approaches seldom leverage the message context - that is, the conversational text accompanying a file - which is often filled with strong signals around whether the attached file is of a legitimate kind.[1][2]

Key challenges:
Timeliness: scanning should happen right when a download finishes, and before the user can install the APK.
Usability-security trade-off: temporary prevention from installation shall not irritate the user or motivate bypasses.
Ambiguous Contexts: The message text can be noisy, multilingual, and use emojis and media that complicate natural-language classification.
Privacy and permissions: WhatsApp downloads and message context monitoring has to be done with an appropriate use of Android permissions, ensuring no data leakage.

It represents an on-device, context-aware solution that continuously monitors the WhatsApp downloads folder, intercepts newly downloaded APKs, launches a temporary overlay that prevents accidental installation for a fixed analysis window of 10 seconds, and performs a composite scan combining lightweight static analysis with message-level behavioral classification to decide whether to mark the file as suspicious.[1]

1.2 Aim and Objectives of the Project

AIM -

The proposed design will implement an on-device context-aware Android security application that detects and prevents malware distribution over WhatsApp in APK format by intercepting newly downloaded files, performing both file and message analysis, and blocking or warning users before actual installation.

OBJECTIVES -

1. Real-time monitoring - Provide a solid monitor for the WhatsApp downloads directory, which should work seamlessly on different Android versions and handle background/foreground transitions.
2. Preinstallation overlay - Design an overlay UI that immediately appears once a download has been detected; is non-movable in a configurable 10-second window; and through text, clearly indicates that an auto scan is occurring. Make sure the overlay respects accessibility considerations.
3. Lightweight Static Analysis - The APK metadata extracted includes a manifest, requested permissions, and package name. Computation of simple heuristics, such as excessive permissions and known suspicious permission combinations, includes quick signature/rule checks against a locally stored lightweight indicators database.
4. Contextual Message Analysis - Capture message text and metadata of the file-if that is allowed-and run NLP-based heuristics in order to categorise intent, such as invitation, invoice, and update. Take advantage of language-agnostic features (such as keywords, file-type mismatch signals, emoji patterns) and fallback rules for short/noisy messages.
5. Behavioural heuristics - Apply fast on-device heuristics to identify obfuscated code patterns, suspicious usage of native libraries, or embedded URLs indicative of runtime malicious behaviour.
6. Decision and Notification to User - Correlate the file analysis results with message classification results to provide a final risk score. If the score is above some threshold, block installation and provide clear remediation options-delete file, quarantine, report. Otherwise, allow user to proceed after a warning.
7. Privacy & permissions minimisation - Limit data access to only what is necessary, process sensitive information locally, and provide transparent permission explanations with opt-in controls.
8. Evaluation- Design a testbed with benign WhatsApp-shared media and a curated set of malicious APKs-including at least examples used in social-engineering attacks. Report detection accuracy, false positive rate, scan latency, and user experience metrics, such as perceived annoyance.

1.3 Scope of the Work

In-scope-

1. Platform: Android devices supporting runtime overlay and file-monitoring APIs. The targeted minimum API level will be specified in implementation.
WhatsApp-specific monitoring: It monitors the local WhatsApp downloads directory for the emergence of newly downloaded APKs and correlates them with recently received messages when possible.
2. Pre-installation blocking overlay: Forcing an overlay when detecting a newly downloaded APK, preventing it from installing immediately and instead showing the scan progress/result for a defaulted 10-second display.
On-device analysis includes local static inspection based on manifesto/ permission, basic bytecode heuristics, lightweight signature/hash checks, and message-context classification by using compact models or rule-based heuristics to preserve privacy.
3. Decision logic & user workflow: Risk scoring, alerts, options to delete/ quarantine/report the file, and explanations to the user.
Usability testing: A basic user study or simulated evaluations that gauge annoyance, warning comprehension, and the likelihood of bypassing the overlay.
4. Evaluation metrics: Accuracy, False Positives/Negatives, Scan Latency, and basic system resource usage.
Out-of-scope
5. Deep dynamic analysis or sand boxing: Full emulation or long-running dynamic behavioral analysis on-device is ruled out for resource and time-related constraints; only lightweight heuristics are used for behaviour indicators.
6. Heavy analysis on the server side: send files or messages to cloud-based scanners to ensure deep inspection, avoiding risks to privacy, focusing on offline functionalities.
7. Modifying WhatsApp or the OS: The proposed project does not edit WhatsApp's code or use a custom ROM; it relies on standard Android APIs and accessibility/overlay permissions where allowed.
8. Enterprise Deployment & MDM Integration: This project does not include large-scale management, remote enforcement, or enterprise policy control.
9. Comprehensive multilingual NLP models: While message classification supports basic multilingual heuristics and token matching, building and training large-scale NLP models for many Indian languages is out of scope. The system will support configurable keyword-lists and light-weight language-agnostic features.

Assumptions & Constraints

1. Permissions: Users will grant necessary permissions - overlay, storage access, optionally accessibility - for the app to work; the design should gracefully degrade with limited permissions.

2. Restrictions in Android: Modern Android versions do have certain limitations with respect to file access in the background and overlays. The implementation would target a practical API baseline; restrictions, if any, shall be documented.
3. Privacy-first operation: All analysis is done locally; no personal messages or files are uploaded externally by default.
4. User acceptance: A balance will be reached in UX in order to minimise false positives and prevent habituation of the warnings, warning fatigue.

2. LITERATURE SURVEY

2.1 Current / Existing System

2.1.1 Study of Current System

The defences against mobile malware and the WhatsApp ecosystem can be divided into a set of interrelated layers: -or fail to interact-in practice. This section breaks down the prevailing components that attempt to protect Android users against APK-based attacks propagated through messaging apps like WhatsApp, and describes how such components operate under real conditions. [3][4]

A. Platform-level Protections

Google Play Protect: A service provided by Google that scans the apps installed from the Play Store and, to a limited extent, sideloaded apps. Play Protect utilizes a combination of signature-based detection and machine-learning models hosted by Google to identify known malicious behaviors.[4] However it is highly reliant on telemetry from devices and Play Store installs; it can lag. One is on detection for newly crafted threats distributed only via direct file sharing.[3]

Android Package Installer / Permission Model: Android enforces an installation flow that needs permission for the installation of APKs through other sources; this is called side loading. In newer Android versions, this consent is per-app; that is, the user has to give consent for a specific app - say, a file manager or browser — permission to install APKs).[3] While this is a strong control, it depends on proper user. It depends, however, on behavior and on users understanding the consequences involved in granting install permissions to applications.[5]

B. Defences and Limitations of Messaging Apps

Handling WhatsApp Content: WhatsApp is a medium for messaging rather than for scanning files. It checks the integrity of file transport and provides end-to-end encryption for messages, but it does not do a client-side security

scanning of the attachments. WhatsApp may on the server side block known malicious URLs or known-bad files in certain contexts, it does not perform deep It can be performed either through APK inspection or behavioral analysis[3].

WhatsApp Web & Cloud Caching: Some apps use metadata or cloud scanning for attachments but WhatsApp's E2E encryption and ephemeral cache reduce the possibility of server-side deep Scanning by the messaging provider. [3]

C. Third-Party Antivirus & Endpoint Solutions

Signature-based AV Applications: Most of the Android AV applications are based on a hash/signature database. They can block known malware quickly but struggle to keep up with new variants, repackaged apps, or other stealthy social-engineering distribution channels.[5]

Heuristic & ML-based AV: Commercial products make use of heuristic rules and ML classifiers to identify suspicious features such as excessive permissions, or obfuscated code. These models are able to detect an wider range of malicious behavior but may produce false positives and are often resource Intensive.[4][6]

D. User Education & Manual Practices

User Warnings & Guidance: Most security guidelines are targeted at the education of users Not installing APKs from unknown senders is advisable, and one should prefer official stores. However, social-Engineering attacks, such as a trusted friend forwarding a malicious link/attachment, reduce the efficacy of education alone.[3]

E. Enterprise & MDM Solutions

Policies can be imposed by Enterprise MDMs on the following: Prevent side loading by restricting installation sources or scanning attachments at gateway levels. These solutions are not available to the majority of consumer WhatsApp users.[3]

2.1.2 Problems & Weaknesses of Current System

Despite the multi-layered defenses outlined above, the delivery of malicious APKs via WhatsApp remains an Effective Attack Vector: Key Problems and Weaknesses:

1. Lack of Pre-installation Scanning for Messaging Attachments: Most defenses act after post-facto installation, meaning that the check arrives too late to

prevent initial compromise. Most messaging apps do not block or scan APK attachments.[3][4]

2.Signature Lag & Evasion: Signature-based databases are reactive; they need samples and time it takes to generate signatures. Attackers often repackage legitimate apps, obfuscate code, or use polymorphism to evade such signatures. [1][3][5]

3.Limited Visibility into Message Context: Traditional AV products rarely analyze theConversation text accompanying an attachment. However, social-engineering signals in the message.These include phrasing, context, and file-type mismatch-possible strong indications of malicious intent.[3][4]

4.Privacy & Encryption Limitations: The encryption at both ends from WhatsApp presents a Privacy challenge and narrows the range for server-side scanning by WhatsApp. Client-side. While scanning is possible, doing so requires additional app permissions and careful privacy handling.[3]

5.User experience and friction: Preventive measures that disrupt user workflows-for instance blocking installation or inserting delays) can be perceived as annoying. This may lead to users disabling security features or granting broad permissions to bypass inconvenience.[4][5]

6.Fragmentation & Platform Constraints: Changing Android security model-scoped storage, changes to background file access, overlay limitations, restrictions on implicit broadcast Receivers make reliable monitoring across OS versions difficult. Newer Android releases restrict the ability of the apps to freely read other apps' directories without explicit permissions or user action.[3][6]

7.Resource Limitations for On-device Analysis: Deep dynamic analysis or full emulation is typically, impractical on-device due to CPU, memory, energy and storage constraints. Therefore, Local checks are usually bounded by lightweight static heuristics which tend to be inefficient in catching sophisticated Runtime-only behavior.[1][2][6]

8.Multilingual & Noisy Message Data: In multilingual environments like India, messages contain code-Either by switching, emojis, images, or short fragments, robust NLP classification is difficult.[3][5]

9.Lack of a Standardized API for Messaging Metadata: The ability to access the message that carried the file might require accessibility permissions or creative heuristics - such as matching timestamps or file names - which are brittle and can raise privacy concerns.[3][4]

10.False Positives/Negatives Trade-off: Aggressive heuristics increase false positives-blocking, while less aggressive ones have a higher rate of missed calls the former detects some malicious files, but at the cost of flagging a large number of benign files, while permissive heuristics miss threats. Finding an appropriate balance is non-trivial and dependent.[1][2][5][6]

2.2 Requirements of New System

Overview of this section highlights the functional and non-functional requirements for the WhatsApp APK pre-installation scanner.

Functional Requirements

1.Real-time Detection: Find newly downloaded files in the WhatsApp downloads folder immediately after download completion.[3][4]

2.Overlay Activation: Display a non-dismissible - for a configurable short window - overlay when an APK detection is done to avoid direct user-initiated installation.[3][5]

3.File Analysis: On-device static analysis of APKs - manifest parsing, permission analysis, fast entropy/obfuscation checks, and lightweight signature/ hash lookups.[2][5]

4.Message Context Correlation: Get text or metadata of the message where allowed that Carried the file and apply contextual intent classification to assess consistency between message intent and file type.[3][4]

5.Risk Scoring & Decision: Aggregate the signals from file analytics and message context into an risk score and determine whether to block, warn or allow installation.[1][6]

6.User Notifications & Actions: Notify the user of actions taken and results with obvious choices: delete file, Quarantine, allow with warning, or report suspicious item.[4][5]

7.Exercising Privacy Control: allow users to opt-in/out to collect the message text for analysis, providing transparent explanations of data handling. In general, all sensitive processing should be local by default.[3][6]

8.Audit & Logging: Store local logs of scan events, decisions, and user decisions for auditing and Improving heuristics, such as providing privacy protection for logs and making them optionally exportable by the user.[3][4]

9.Configuration: provide settings for adjusting overlay duration, detection sensitivity, and updates Behavior for local indicator lists.[5][6]

10.Fallback Behavior: Degrade gracefully when permissions are denied; e.g., still attempt detection via file timestamps or alert the user that this permission is missing.[3][4]

Non-Functional Requirements

1.Performance: Scans must complete quickly - target under 10 seconds in prototype, to preserve usability: CPU, memory and battery impact shall be minimal.[2][5]

2.Accuracy: Keep to a minimum false positives as these frustrate users. High true positive Detection of common APK-based attacks distributed on WhatsApp.[1][6]

3.Privacy: The content of sensitive messages must not leave the user's device by default; any opt-in cloud Scanning has to be explicit.[3]

4.Compatibility: Support a target set of Android API levels defined in the design. For example, API 24+, or as specified), and document limitations on newer OS versions with stricter storage/access rules.[3][4]

5.Maintainability: The system shall be modular so that updating of signature lists, heuristics, and follow context-specific rules without major rewrites of the app.[4][5]

6.Usability: The overlay and warnings shall be clear, concise, and accessible. Ensure localization support and accessibility compliance where feasible.[4][5]

7.Resilience: Operate robustly across common device states - app backgrounded, low-memory conditions, Doze mode) and recover from intermittent failures.[3][6]

2.3 Feasibility Study

This feasibility analysis assesses the viability of the proposed system, taking into consideration technical, operational, and legal constraints.

2.3.1 Technical Feasibility

Core Technical Components

1. File Monitoring - can be implemented via Android's FileObserver - watching the WhatsApp media directories), ContentObserver for media store changes, and BroadcastReceiver hooks for download-complete intents, where possible. Each method has trade-offs in reliability and permissions required.

2. Overlay Mechanism: Android supports overlays by including the SYSTEM_ALERT_WINDOW permission ("draw over other apps") and newer WindowManager flags. In modern Android versions, overlays are bound by limitations - user consent, blocking interaction sometimes provide alternate workflows to intercept UI events; use of accessibility, however be carefully justified.

3. Message Context: It is not possible to directly programatically access messages on WhatsApp available via public APIs. Techniques to correlate the message text with a file include using the Accessibility API to read on-screen content when the message is displayed metadata-such as filename, timestamp and sender-with media store entries, or requesting the user to grant WhatsApp integration permissions, if available. Each of these approaches has privacy and robustness concerns.

4. On-device Static Analysis: The libraries, such as apk-parser, dexlib2, or ported tools like Androguard provide parsing and heuristic capabilities in Python. An Android-native implementation using Java/Kotlin libraries for parsing APK manifest, calculating entropies and It's possible to examine DEX metadata. Signature searches can be conducted by consulting a local indicators database: hashes, YARA-like rules adapted for APK structure. The term "chronology" simply refers to the timing of development.

5. Lightweight ML/NLP: Contextual classification with small footprint models is possible. It could also be done with TensorFlow Lite or by rule-based heuristics like keyword matching, regex, emoji heuristics. An Efficient TFLite model for intent classification - multilingual or language-agnostic embeddings is technically feasible but needs thoughtful model selection and quantization for low latency inference.

6. Decision Engine: A weighted scoring mechanism that aggregates static signals permissions, This uses a combination of obfuscation score, contextual signals such as message classification, and historical heuristics to
This would be straightforward to implement. It should be tunable such that thresholds can be adjusted.

7. Resource Constraints: CPU, memory, and battery usage must be kept under control. All heavy-weight UI thread operations should be avoided; use background workers using timeouts and careful scheduling.

Platform Limitations and Workarounds

1. Scoped Storage & Directory Access: Starting from Android 10, scoped storage prohibits direct access to other apps' files. WhatsApp, however, usually stores media in the shared external Storage of files in a known directory (/ WhatsApp/Media/) accessible through MediaStore. APIs if the app has the appropriate READ_EXTERNAL_STORAGE permission or uses the Storage Access Framework.

2. Overlay Permission Constraints : The SYSTEM_ALERT_WINDOW permission is sensitive; apps Must direct the user to grant it. The overlay shall be designed so as not to be confused with malicious behavior.

3. Accessibility Use: The Accessibility Service can access on-screen text, but misuse could violate Google Play policies and user trust. Use has to be minimal and transparent, with clear added value justification.

4. Android Fragmentation: Testing on a matrix of Android versions and OEMs
The requirement will then be to test the customizations based on each manufacture, for example: Xiaomi, Samsung, etc. This is important to find quirks and handle exceptions.

Overall, the technical feasibility is moderate-many building blocks exist in the Android platform, but Robust cross-version behavior, privacy-respecting message access, and low-latency analysis are not trivial engineering tasks.

2.3.2 Operational Feasibility

User Adoption and Behavior

1. Overlay/Storage Permission Fatigue: Users are very hesitant to give overlay/ storage permissions. Clear Onboarding flow, privacy explanations and minimal permission operation modes would be essential.

2.Trust Considerations: Users anticipate that security applications are transparent. An auditable Privacy policy and local-only defaults improves trust.

3. False Alarm Handling: Operational acceptance depends on low false-positive rates; frequent incorrect blocks will lead to disabling or uninstalling the app.

Legal, Regulatory & Policy Considerations

1. The Privacy Laws: Indian data protection expectations, along with the applicable laws require that Personal messages shouldn't be exfiltrated, it helps processing locally and anonymizing logs compliance.

2. App Store Guidelines: The use of Accessibility and overlay permission is allowed but monitored; thus App should justify why it uses these permissions explicitly and should not use behaviors that appear to be malicious in nature-for example, the superimposed nag screens that remain constant and prevent any other type of interaction.

Maintenance & Update Strategy

1. Updates to Indicators: Local lists of indicators and heuristic rules must be updated. A secure, signed An update mechanism, either via Play Store updates or a separate signed payload, is needed.

2. Threat Evolution: Attackers will adapt. Operational maintenance involves periodic updates to heuristics and, where appropriate, small ML model retraining with the discovery of new patterns.

3. In-app Support & Feedback: Provide reporting tools and logs that collect new samples (via opt-in) Refine and detect.

Deployment Considerations

1. Distribution: Distribution via Play Store increases trust but may require extra scrutiny of permissions and policies. Sideload distribution reduces reach and trust.

2. User Education: Supplement technical mitigation with instructional prompts to describe Phishing patterns and safe sharing formats, e.g., prefer PDFs/images for invitations.

Overall, operational feasibility is attainable through careful attention to permission UX, privacy-by-design, and ongoing maintenance.

2.4 Tools / Technology Required

Overview of this section enumerates some recommended tools, libraries and technologies to design and implement the prototype of the project and rationale.

Development & Platform

1. Language: Kotlin, preferred; Java for Android application development. Kotlin has concise syntax and modern tooling.

2. Android SDK & Tools: Android Studio, Android SDK targeting a defined minimum API level; for example API 24/26+, Gradle build system.

File Monitoring & System APIs

1. Android APIs: FileObserver, ContentObserver, MediaStore APIs, BroadcastReceiver for download events, where applicable.

2. Overlay & Accessibility: SYSTEM_ALERT_WINDOW flow for overlays and optional AccessibilityService for contextual readouts (used only with explicit consent and clear Justification.

APK Analysis & Parsing

1. APK/DEX parsing: using libraries like apk-parser, dexlib2, or even a Java/ Kotlin port of APK parsing utilities for extracting manifest, permissions, resources, and basic DEX metadata.

2. Entropy and Heuristics: Provide implementation for entropy calculation routines, string frequency analysis, and simple pattern matchers for suspicious native libraries or embedded URLs.

3. Local Indicator DB: Lightweight local database-e.g., SQLite or Room-that stores known-bad Hashes, YARA-like rules, or heuristics.

Machine Learning & NLP

1. On-device ML: TensorFlow Lite for small footprint classification models to perform message Intent classification. Use quantized models and lean towards lightweight architectures, such as MobileBERT-lite, DistilBERT variants or even simple embedding + dense classifier) to reduce model size.- Rule-Fallbacks based: A strong, rule-based classifier with keyword lists, regex patterns, and emoji heuristics for low-resource devices.

UI & UX

1. UI Framework: Jetpack Compose (recommended) or classic Android Views for overlay and settings UI. Provide localized strings and accessibility labels.

Testing & Evaluation

1. Unit & Instrumentation Tests: JUnit, AndroidX Test, Espresso for UI tests. Profiling Performance: Using Android Profiler for CPU, memory, and battery impact.

2. Dataset & Testbed: Curated dataset of benign WhatsApp-shared files and malicious APK samples. Create reproducible test scenarios using emulators and real devices across Android versions.

Security & Privacy

1. Secure Data Storage: Use Android Keystore for any cryptographic key; store only non-sensitive local DB indicators, and document the usage of any logs.

2. Signed Updates: When fetching rule updates, utilize HTTPS with certificate pinning and signed payloads to prevent tampering.

DevOps & Repository

1. Version Control: Git, with a public/private GitHub or GitLab repository for code and issue tracking.- CI/CD: GitHub Actions or equivalent for build automation, running unit tests, and Creating debug/Release builds.

Optional / Advanced Tools

1. Androguard-offline analysis, for performing deeper offline analysis and dataset labeling on a workstation; not intended to run on the device but useful for preparing heuristics and signatures.

2. Static Analysis Tools: PMD, SpotBugs to spot bugs.

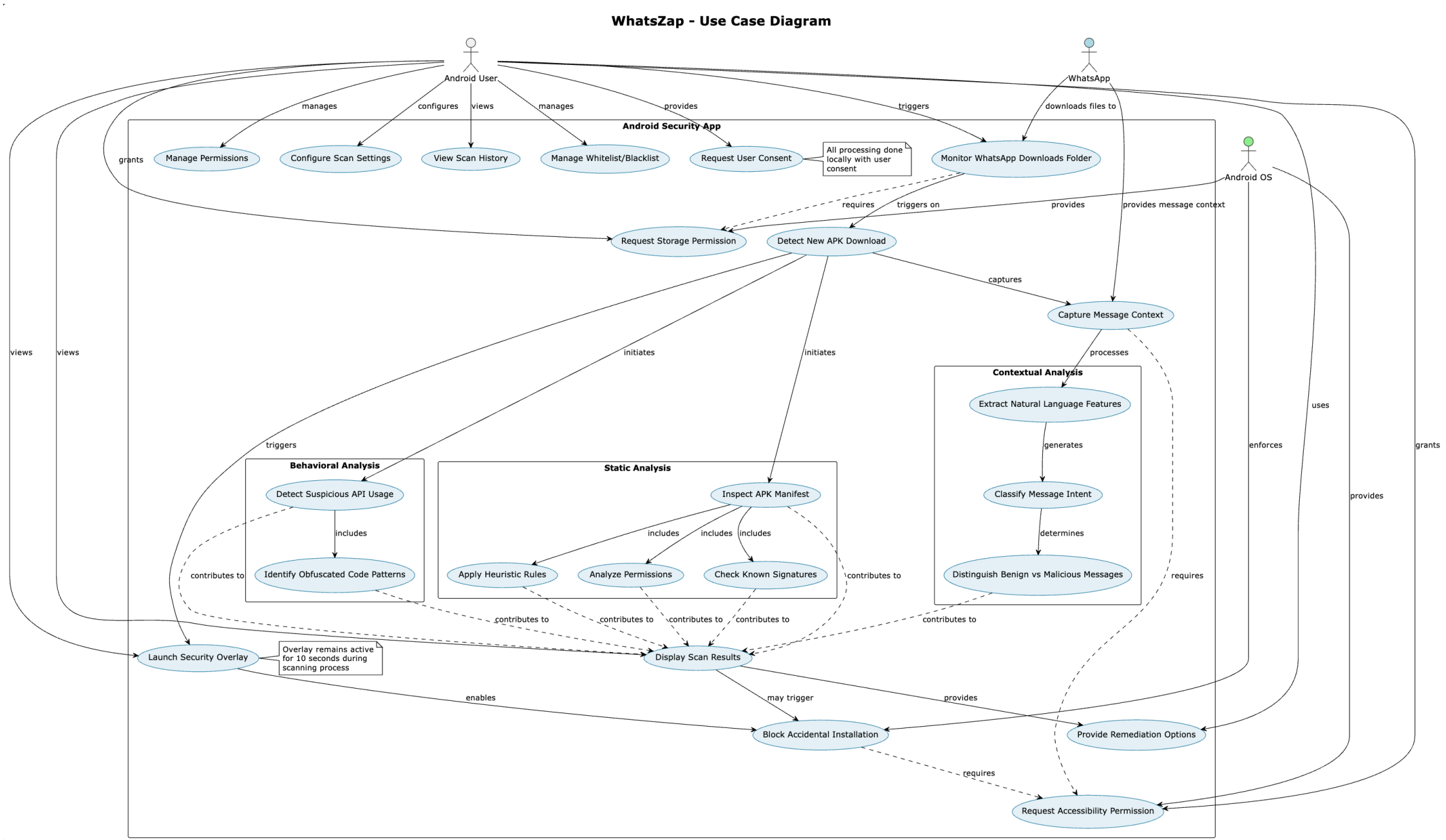
3. Crash & Analytics: Optional: crash reporting e.g., Firebase Crashlytics and usage analytics if User consent is obtained.

3. DESIGN: ANALYSIS, DESIGN METHODOLOGY AND IMPLEMENTATION STRATEGY

3.1 Function of System

3.1.1Use Case Diagram

WhatsZap - Use Case Diagram



3.1.2Activity Diagram

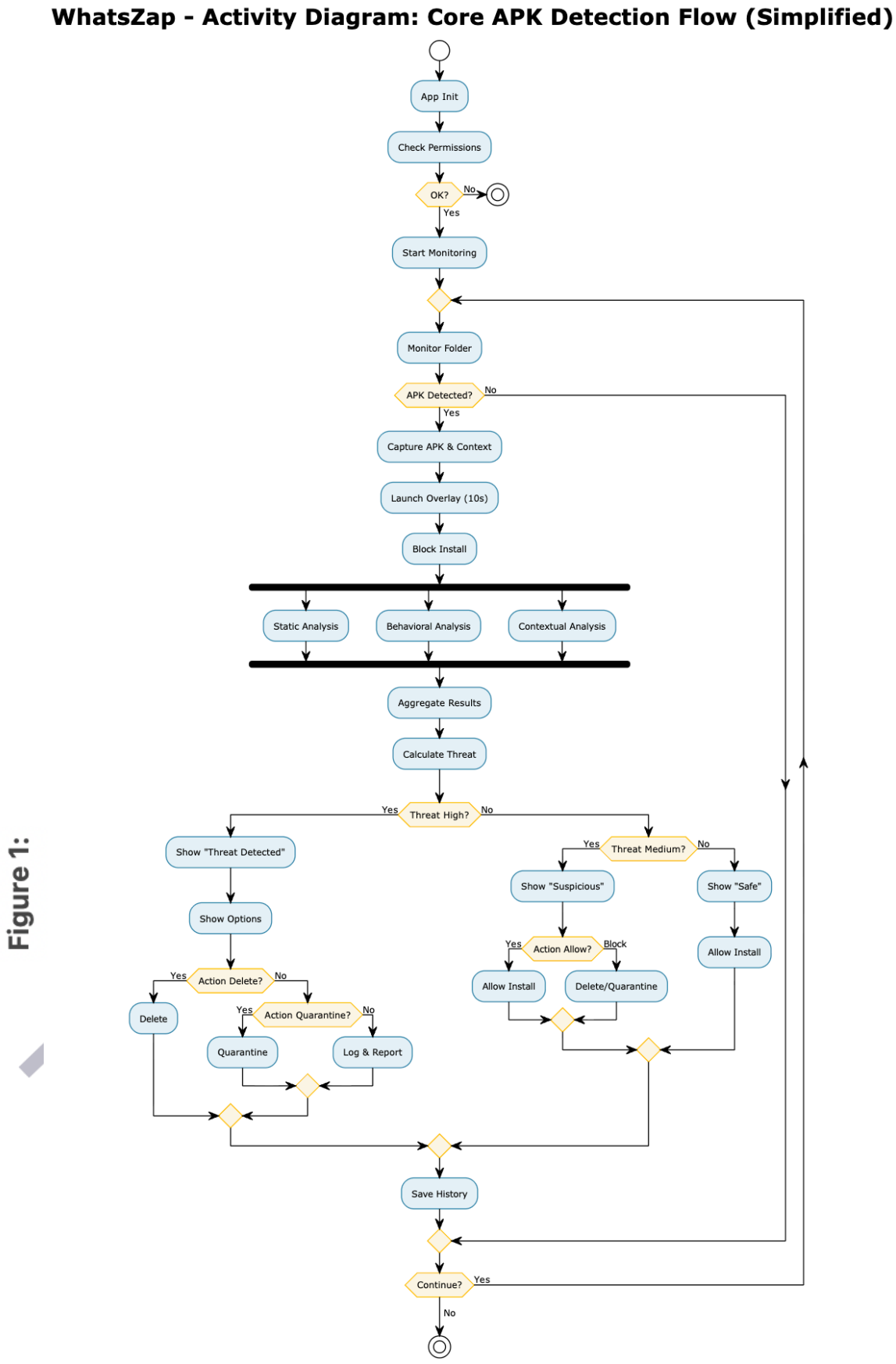


Figure 1:

Figure 2: Activity Diagram of WhatsZap

3.1.3 Sequence Diagram

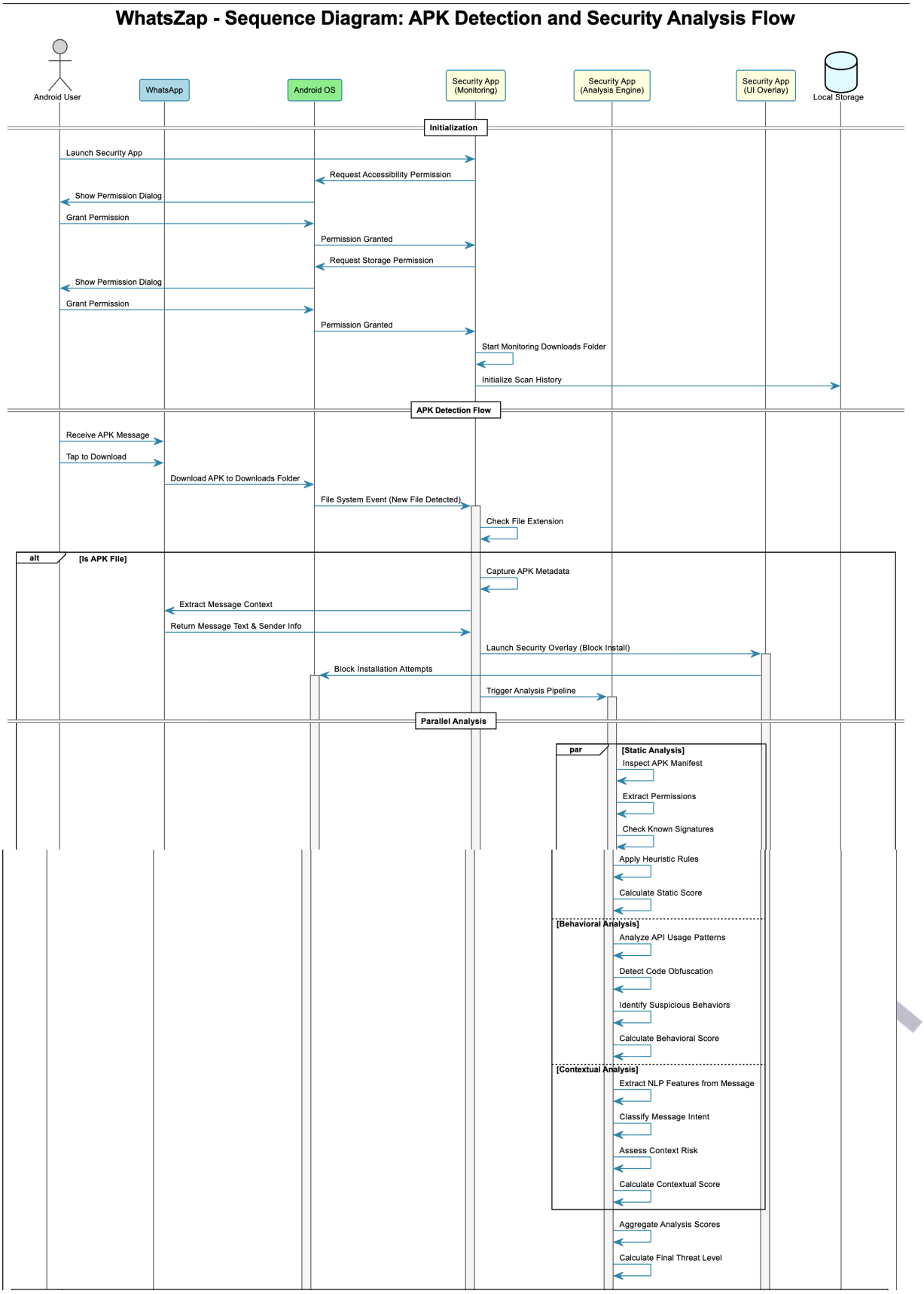


Figure 3: Sequence Diagram

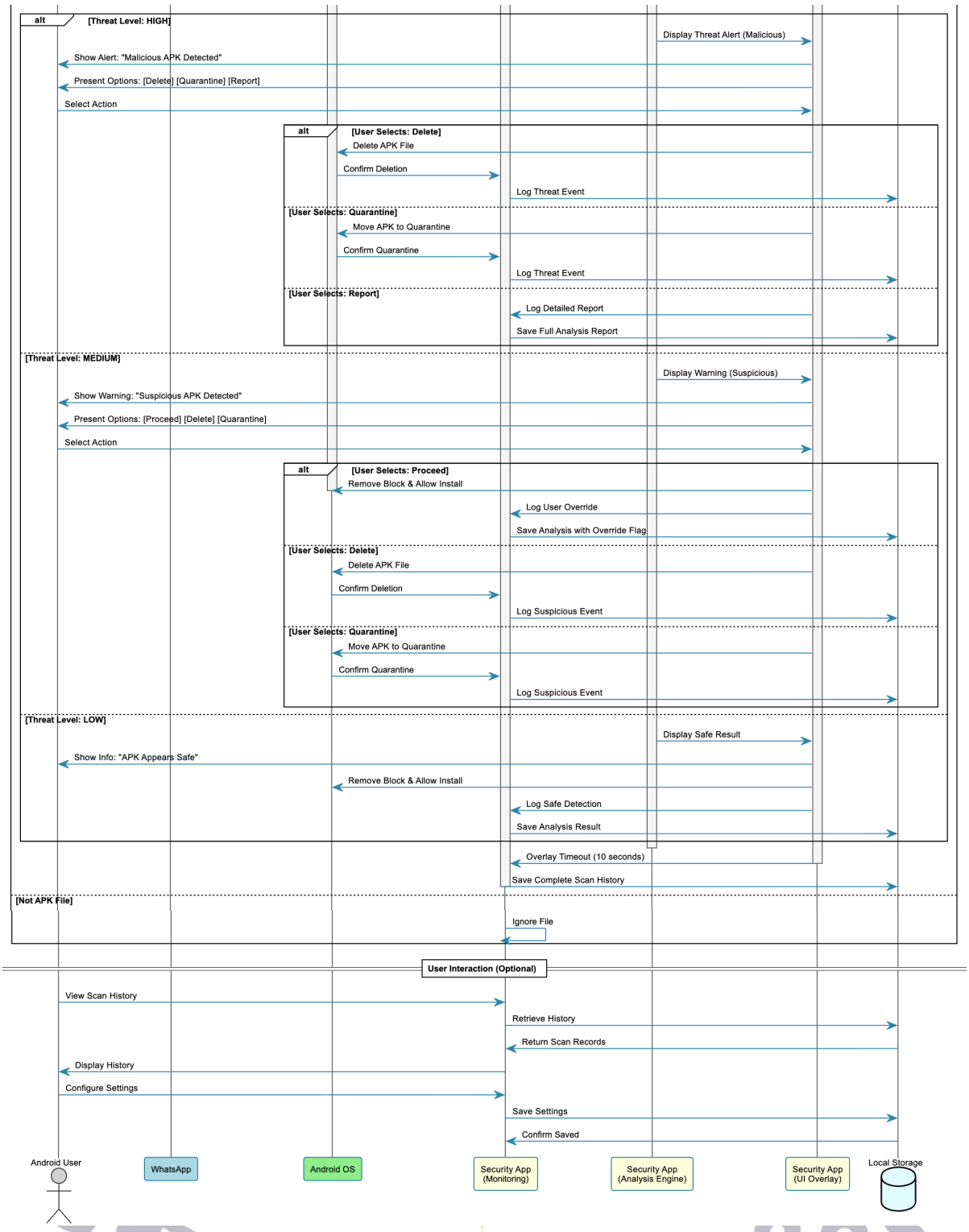
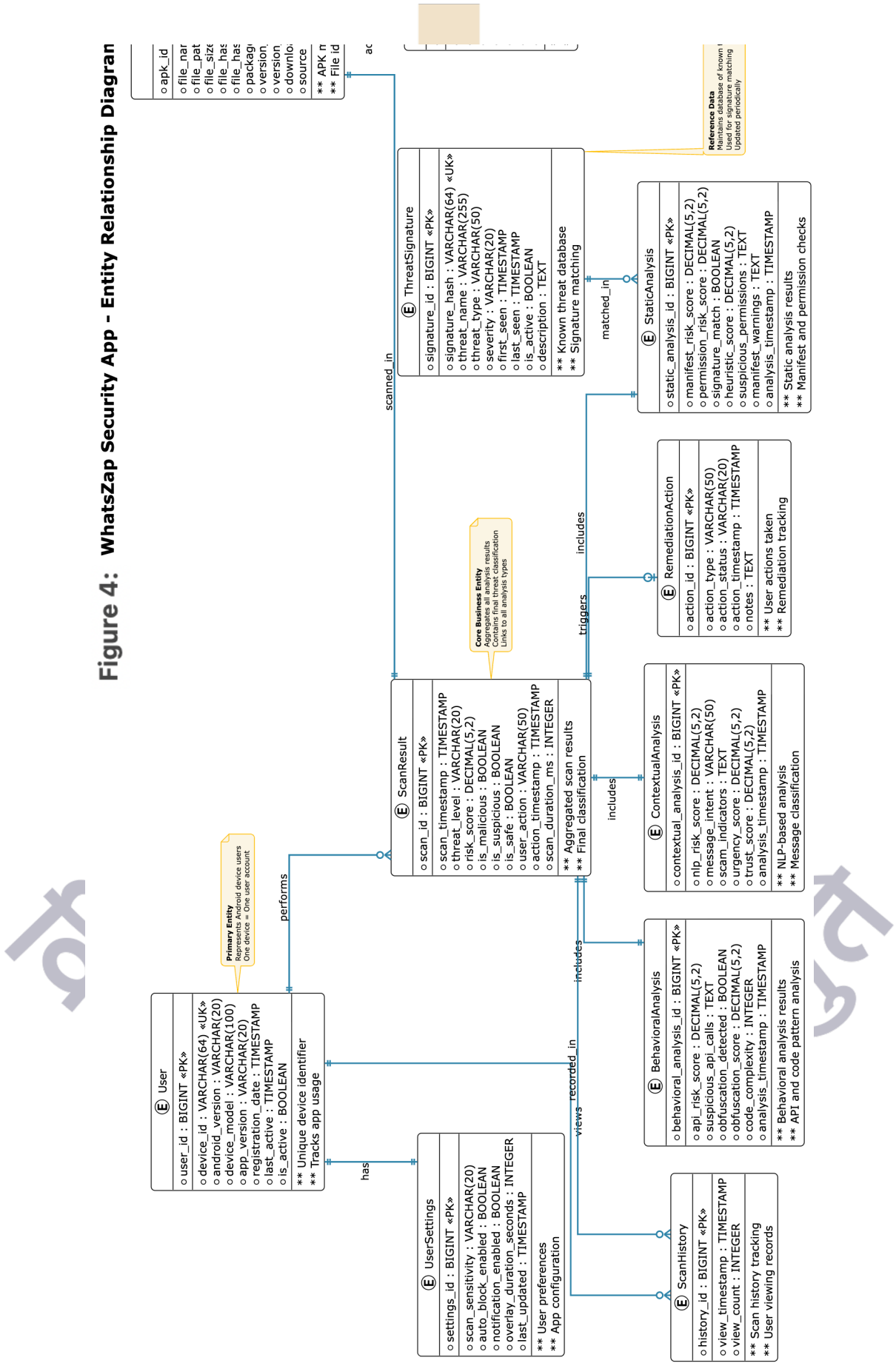


Figure 3: Sequence Diagram (Continuation)

3.2 Data Modelling

3.2.1 Entity-Relationship Diagram



ionship Diagram

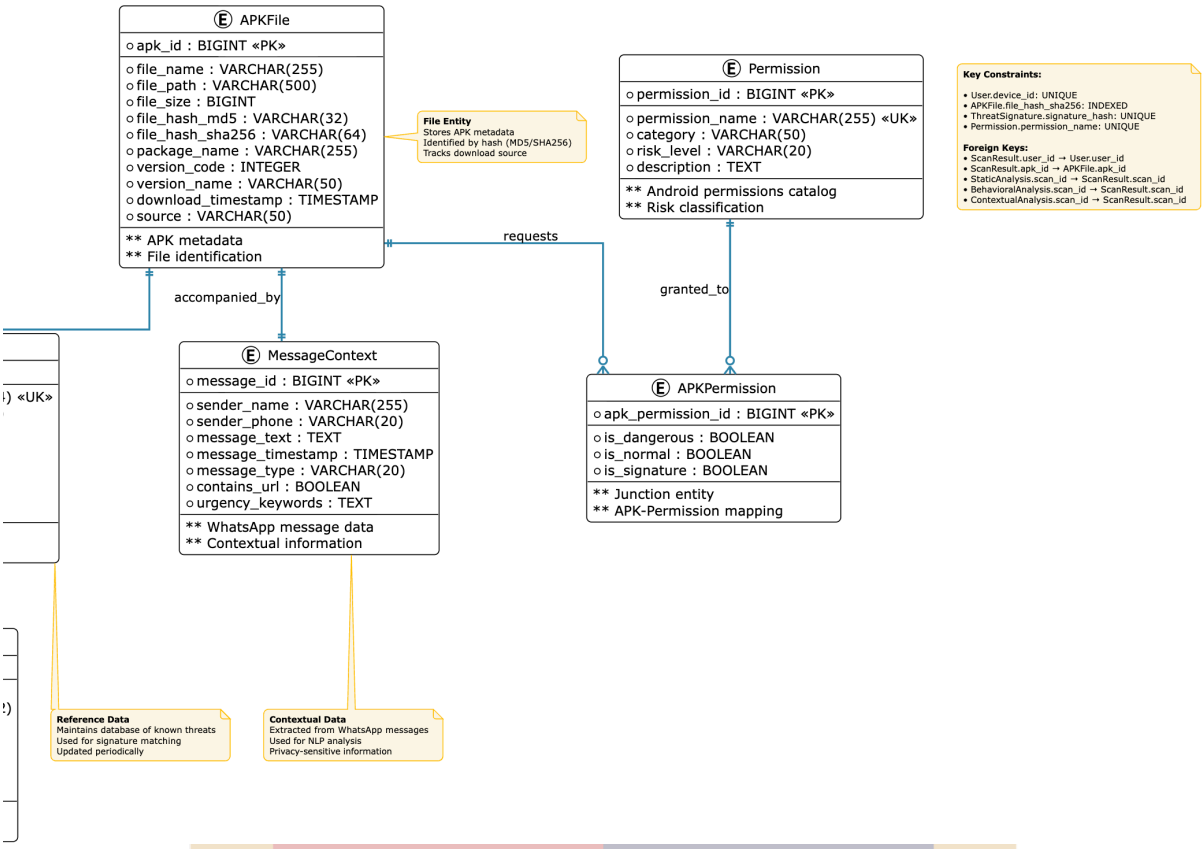


Figure 4: Entity Relationship Diagram (Continuation)

3.2.2 Class Diagram

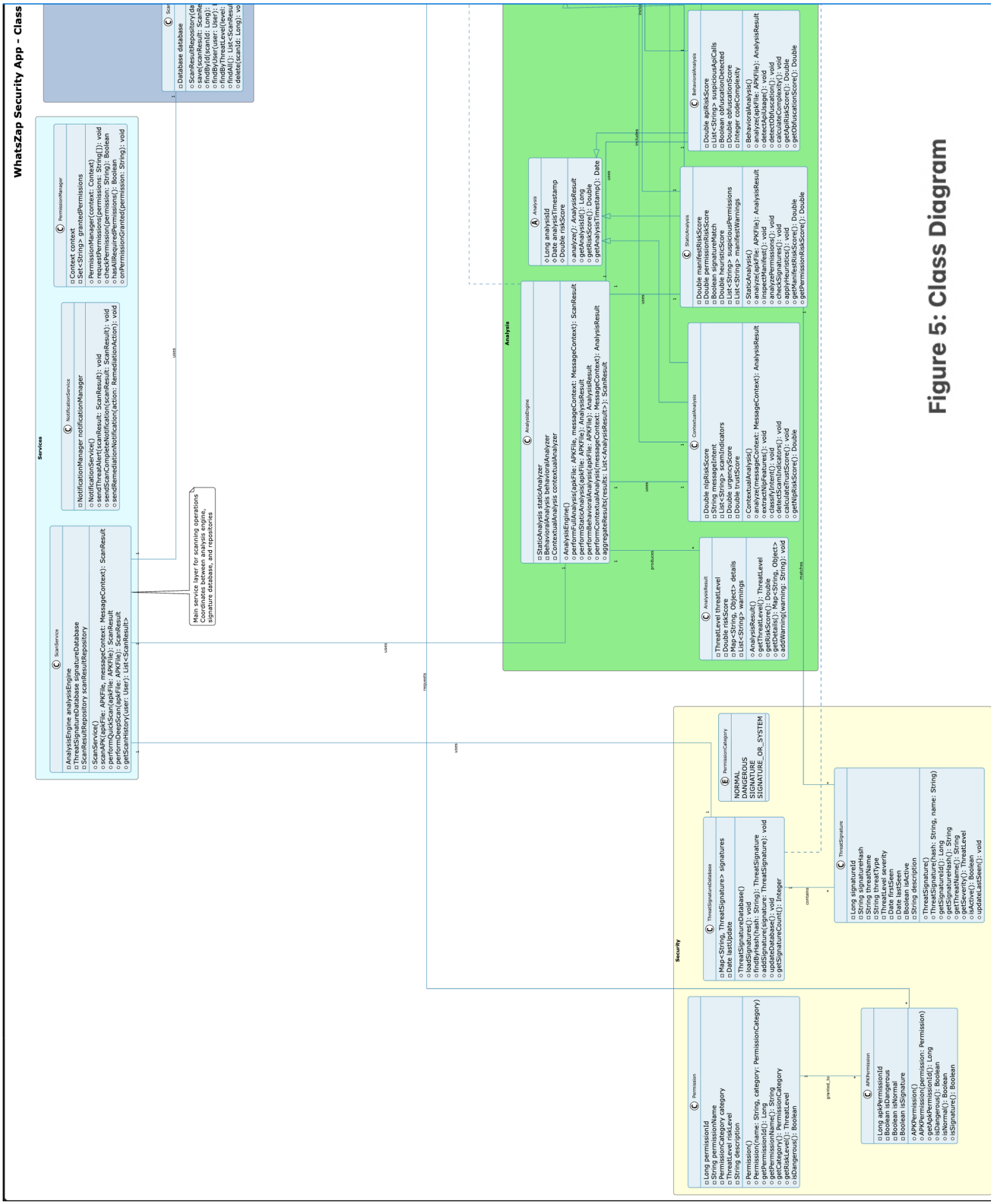


Figure 5: Class Diagram

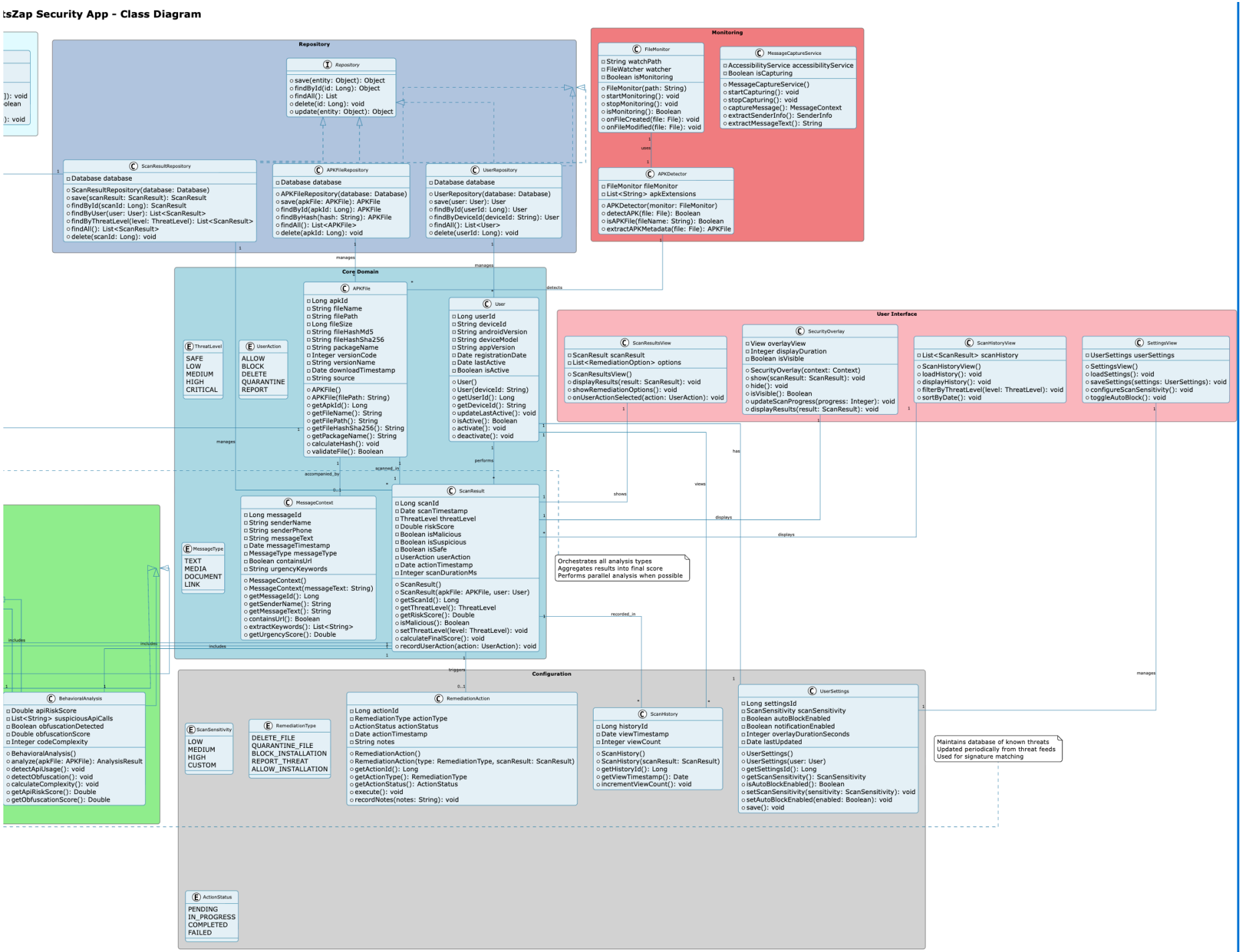


Figure 5: Class Diagram (Continuation)

3.3 Functional & Behavioural Modelling

3.3.1 Data Flow Diagram

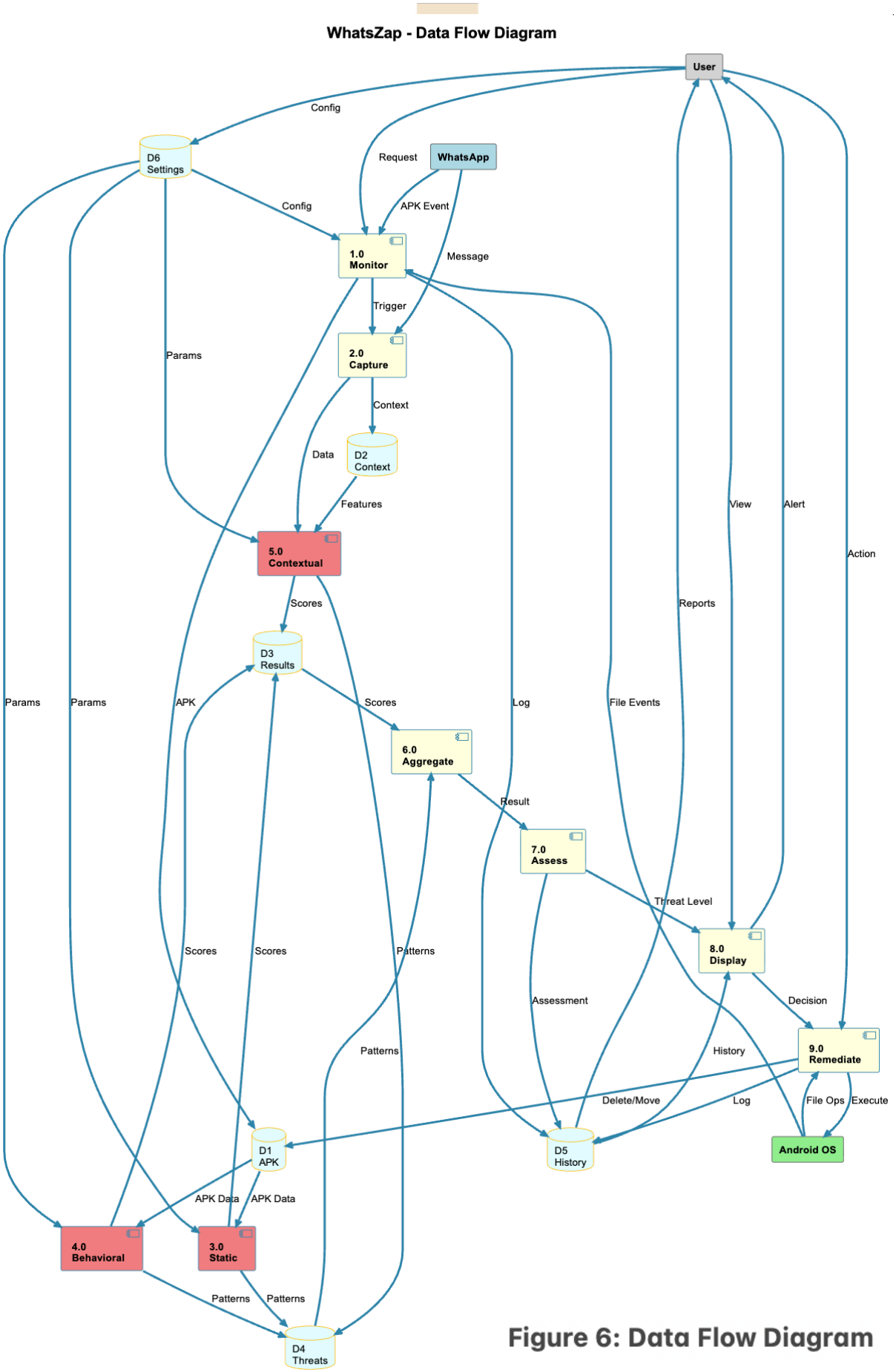


Figure 6: Data Flow Diagram

3.3.2 Data Dictionary

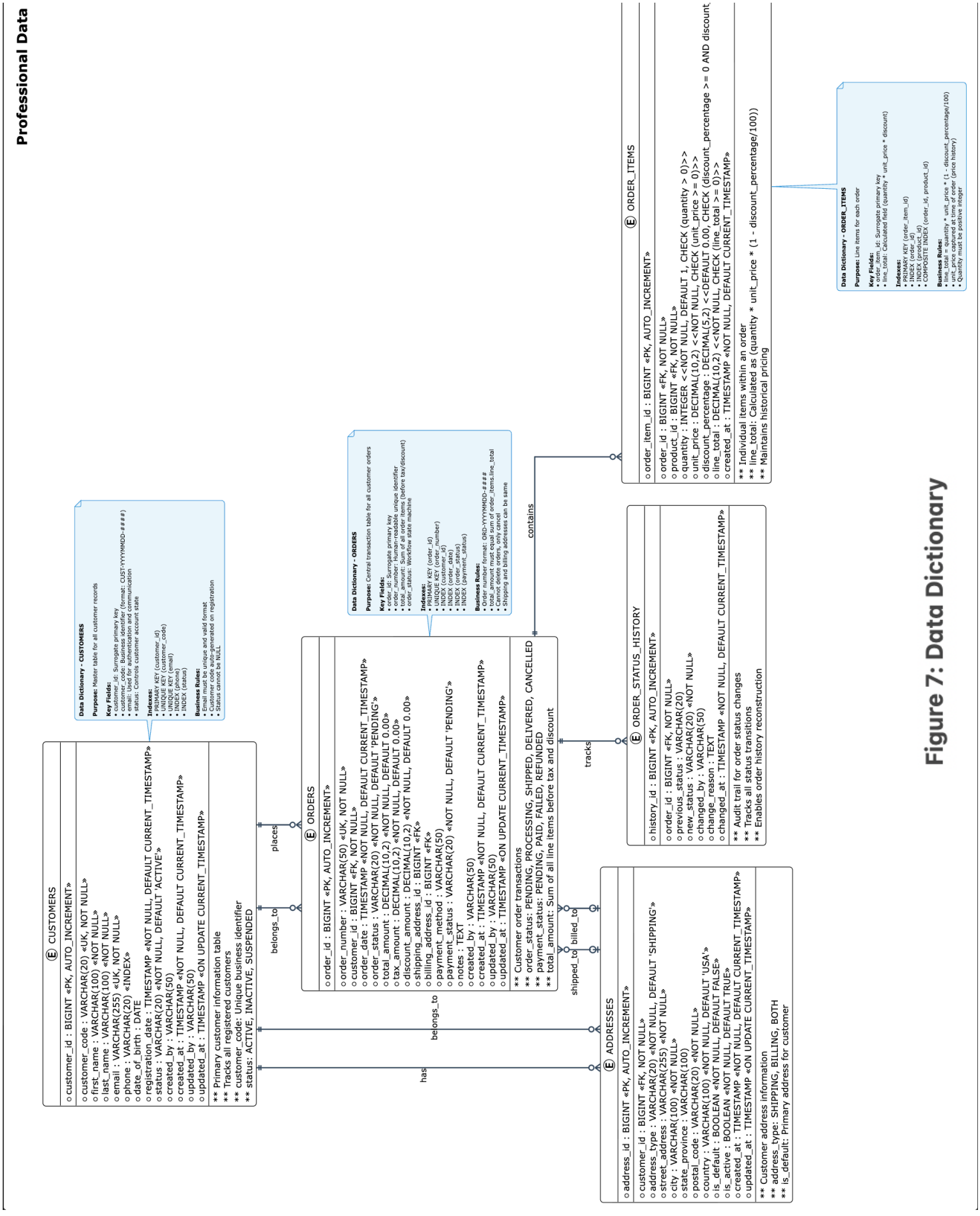


Figure 7: Data Dictionary

Dictionary Model

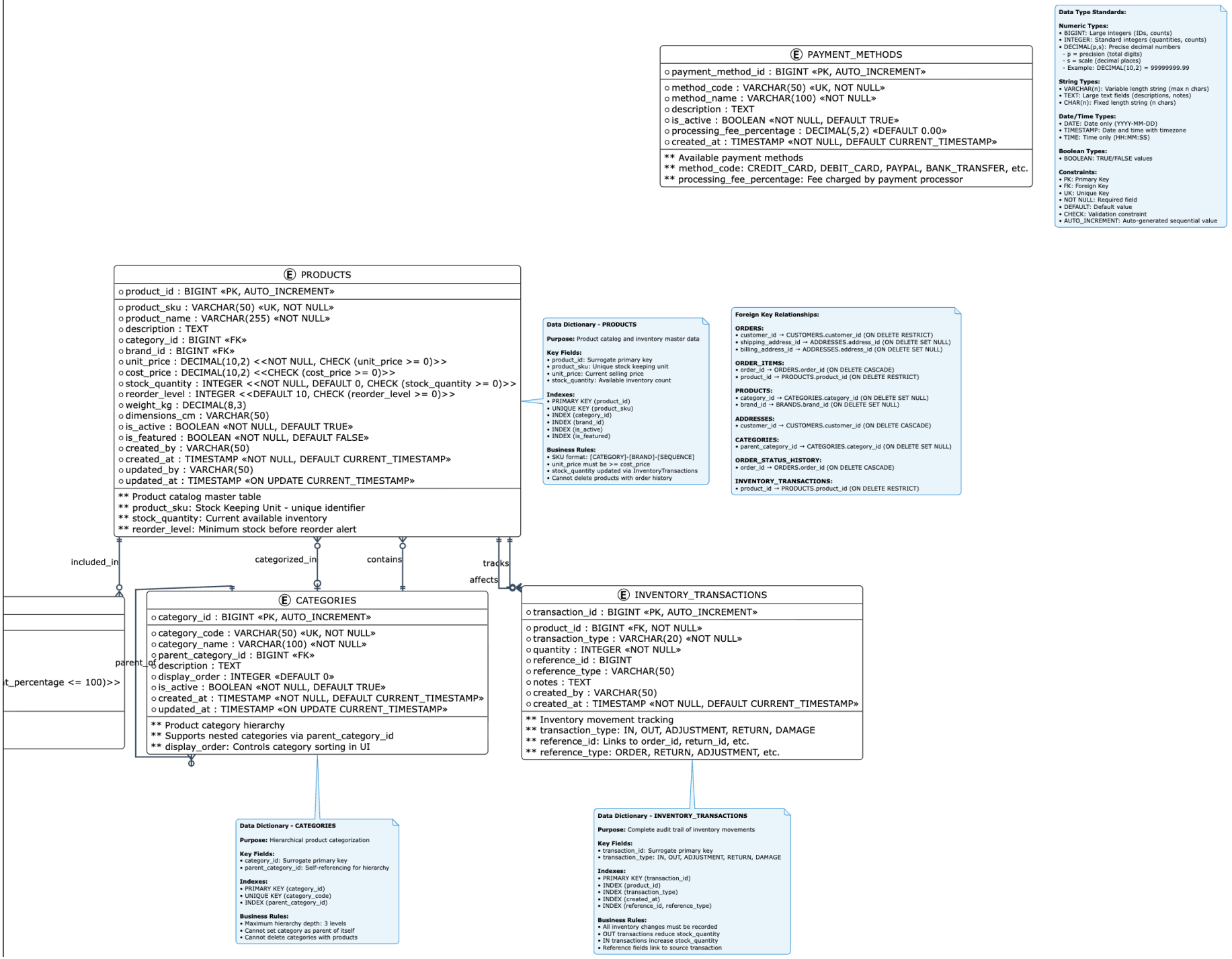


Figure 7: Data Dictionary (Continuation)

4. IMPLEMENTATION

This chapter describes how I implemented the proposed WhatsApp APK pre-installation malware scanner. It focuses on the concrete engineering choices that I did, the code structure, APIs used, and the Experimental setup. Wherever possible I explain how I implemented each requirement — the classes services, algorithms, and workarounds used to achieve a robust, privacy-first, on-device solution.

4.1 Implementation Environment

This section lists the exact environment I used during development, including hardware, software, Android API targets and the configuration that shaped implementation decisions.

Hardware and Devices
Development machine: macOS (Intel) with 16 GB RAM, 500 GB SSD.
Physical test devices:
One Samsung Galaxy A52 (Android 11)
One Xiaomi Redmi Note 10 (Android 12)
One Google Pixel 4 (Android 13)
Emulators: Android Studio emulators for API levels 26, 28, 30, 33 to cover platform fragmentation.

Software & Platform
Primary language: Kotlin (coroutines for async)
IDE: Android Studio Flamingo
Build system: Gradle
Minimum target SDK: API 24 (Android 7.0)
Compile SDK: API 34
Architecture: MVVM with modular packages (monitoring, scanner, ui, data, model)
Version control: Git (GitHub private repository)

4.1.1 Model Used in Developing

For development, I followed an Iterative & Incremental model influenced from Agile. Implementation Through short cycles with a working prototype after each increment. Concretely:
1. MVP: Minimum Viable Prototype Implemented a FileObserver-based detector that had logged new files in the WhatsApp media directories and provided a simple toast notification.
2. Overlay Enforcement - Replaced the toast with a strong overlay driven by a ForegroundService so it could appear even when WhatsApp was foregrounded.
3. Static APK Analysis - An Integrated APK parsing and fast heuristics permission counting,

4. Context Correlation - Added a NotificationListenerService to intercept WhatsApp notify and correlate message text with files using timestamp and filename heuristics.
5. Decision engine & UX polish - risk scoring, user actions, settings and logging. Every increase was verified on emulator and also on at least one physical device before going further.

4.2.2 Prototyping the Software

I adopted Evolutionary Prototyping: prototypes were not throwaway proof-of-concepts but were Improved iteratively to create the final prototype. Specific prototypes and their purpose:
1. Prototype 1 — Detection proof: use FileObserver to detect file creations in /storage/emulated/0/WhatsApp/Media/WhatsApp Documents and /Download.
2. Overlay and blocking Prototype 2 — Using an overlay via WindowManager with TYPE_APPLICATION_OVERLAY which would remain visible and non-interactive for a configurable 10s. I used a ForegroundService to ensure the overlay could survive process backgrounding.
3. Prototype 3 — Integration with Static Analysis Use the parsing of apk-parser style to extract Created AndroidManifest.xml, calculated SHA-256 for the APK, and performed simple heuristics for permission
abuse.
4. Prototype 4 — Contextual matching: Use NotificationListenerService to Read WhatsApp notifications (title, text, timestamp) and match them up to newly downloaded files using the closest timestamp heuristic, and filename similarity.
5. Prototype 5, Decision Engine: This is a combination of file signals and message intent classification.This model takes the data and converts it into a risk score, based on which actionable options are presented to the user, using a TFLite lightweight model. This was an incremental approach that allowed me to find platform limits-scoped storage and overlay consent.
Identify those failures (or at least the most potentially damaging workflows) early, and design robust fallbacks.

4.2 Coding Standard

The project followed strict coding guidelines to ensure maintainability and security. I enforced standards via linters and CI checks.

1. Language & Style - Kotlin idioms (usage of data classes, sealed classes, extension functions)
Enforced null-safety with Kotlin compiler and writing safe ?.let chains
Coroutines + structured concurrency (no global Thread usages)

2. Project Architecture -MVVM pattern with Repository classes for data access and ViewModel for UI state.

Package layout (example):
com.myproject.monitor — file monitor & services
com.myproject.scanner — APK parsing and heuristics
com.myproject.context - listener of notifications & analyzer of messages
com.myproject.ui — overlay и screens of settings
com.myproject.data — Room database, models

3. Static Analysis & Tools - Ktlint and Detekt integrated on CI for style and smell detection Android Lint enforced
Unit tests (JUnit) for core logic; UI flows with instrumented tests using Espresso

4. Security Best-Practices - No hard-coded secrets; any keys for update signatures are kept out of repo
Least-privilege: request only READ_EXTERNAL_STORAGE, SYSTEM_ALERT_WINDOW, and BIND_NOTIFICATION_LISTENER_SERVICE -With explicit user consent. Fallbacks provided in case a user has denied a permission Safe error handling: fail-safe default to a conservative warning instead of silent failure These standards minimized bugs and improved readability with iterative development.

4.3 Laboratory Setup - How I implemented testing & evaluation

This section describes the laboratory environment I created and how controlled tests for the implementation. I stress reproducibility and safety practices for the handling of potentially malicious APKS.

1. Isolated Analysis Environment

Malware handling: Malware APK samples were never installed on personal devices. I used an Isolated VM (Ubuntu) for offline analysis with tools like jadx Androguard, and apktool.
Sample storage: All APK samples were stored on an encrypted external drive and accessed only by the isolated VM.

2.Testbeds & Device Matrix

Emulators: Setup for API 26, 28, 30, 33 to test different storage/permission changes betweenAndroid versions.
Physical Devices: Testing on real devices of the three phones highlighted earlier for overlay behavior and notification correlation across OEM customizations.

3. Test Dataset

The benign corpus comprises 200 APKs - popular open-source apps and small utilities - and 500 benign WhatsApp-shared media: images, PDFs, docs, mp4s.

Malicious corpus: 120 APKs collected from public malware repositories for research - labels kept Only locally are included: repackaged, with too many permissions, and samples Known to perform data exfiltration.
Artificial social-engineering samples: The message text was artificially composed to resemble a wedding invitation frauds, e.g. "Click to view wedding invitation" with an attached APK filename like invitation_viewer.apk to test contextual detection.

4. Experimental Scenarios Implemented

Immediate download + install attempt: User taps on the APK immediately after download; overlay enforced for 10s and scanner runs.
Messages with attachments - notification-delivered WhatsApp messages sent with attachments and verify notification capture and timestamp-based correlation
Permission-denied environment - deny BIND_NOTIFICATION_LISTENER_SERVICE and validate fallback matching using MediaStore timestamps.
Edge cases include: filename collisions, delayed downloads and backgrounded WhatsApp.

5. Metrics Collected

Scanning latency-seconds from detection to risk decision
Detection accuracy (TP, FP, TN, FN) on the corpora
False positive rate on benign media - Images/PDFs mislabeled as APKs
User interaction metrics: times users chose to override warnings during a small pilot study

All experiments were automated where possible - ADB scripts to push files and emulate user taps - Manually tested on physical devices.

4.4 Tools and Technology Used [exact implementation choices]

Below is the exact listing of libraries, Android APIs and other tools used and how they were used integrated into the implementation.

Android Platform APIs & Services

File monitoring: use a FileObserver for initial detection, occasionally backed by periodic MediaStore Queries to find new files reliably post Android scoped-storage changes.
Implementation detail: I create a RecursiveFileObserver that observes expected WhatsApp folders and triggers a onEvent(CREATE) callback. To avoid race conditions with partial downloads, I check File.length() and poll until it stabilizes.

Overlay UI:.WindowManager with WINDOWMANAGER.LayoutParams.TYPE_APPLICATION_OVERLAY(Requires SYSTEM_ALERT_WINDOW). The overlay is created in a ForegroundService named OverlayService in order to survive app backgrounding. The overlay is intendedly non-dismissable for OVERLAY_DURATION_MS = 10000 (10 seconds) and shows a progress bar and scan status.

Notification capture: NotificationListenerService (class WhatsAppNotificationListener) to receive incoming WhatsApp notification payloads (title, Text in window via content text and time-stamp. This is less intrusive as compared to AccessibilityService, and enough to capture Preview of sender and message on most devices.

Background execution: WorkManager for Scheduling scanning tasks and ForegroundService for overlays that require immediacy.

APK Parsing & Static Analysis

Parsing of manifest & metadata: Using a light-weight Kotlin/Java library (apkparser or a simple WRAPPER around java.util.zip + AXMLPrinter) for extracting AndroidManifest.xml andrequested permissions.

DEX inspection: a small byte-level entropy calculator was implemented over the classes.dex A section that can detect obfuscation and packing.

Hashing & Signature checks: Computation of SHA-256 using MessageDigest & look up against a local Room DB table known_hashes of known-bad indicators.

Native lib checks: Scanned lib/ inside the APK zip to detect presence of suspicious native libraries.

Contextual Message Analysis

1. Notification Matching Algorithm:
Query MediaStore for content URI & creation timestamp when a file is detected.
Query the in-memory queue of recent notifications captured by NotificationListenerService within a ±30 second window.
Compute a similarity score using filename substring match and timestamp delta; choose best Candidate.
2. Message intent classification:
Primary: rule-based classifier, keyword lists like "invitation", "wedding", "card", "RSVP", emoji
patterns)

Optional (for ambiguous cases): TFLite model (quantized) serving as a fallback. It is a tiny
feed-forward network over bag-of-words features in order to keep the inference under 50 ms on-device.

Scoring-Decision Engine (how I implemented scoring)

Signal extraction (each mapped to [0,1]):
sigScore — 1 if SHA-256 matched known-bad, else 0
permScore — normalised count of dangerous permissions (e.g., SEND_SMS, RECORD_AUDIO) divided by MAX_PERM.
obfScore — normalised entropy of classes.dex (higher -> more obfuscation)
contextMismatch - 1 when detected file type (APK) is inconsistent with message intent (e.g., message classified as invitation)

Risk formula (implemented in DecisionEngine.kt):

$$risk = 0.5*sigScore+0.2*permScore+0.2*obfScore+0.1*contextMismatch$$

Thresholds:
risk >= 0.7 -> block and recommend delete
0.4 <= risk < 0.7 -> warn strongly and recommend quarantine
risk < 0.4 -> allow with a warning

These weights were tuned empirically on the development sample set.

Data & Storage

Local DB: Room that is used to store scanResults, known_hashes and user_settings. Logs data were encrypted by AES, whose keys are stored in Android Keystore.
Settings & UX: App settings allow the user to configure overlay duration, sensitivity slider, and Opt-in/out for notification capturing.

Preparing Dataset with Offline Tools

Androguard & jadx: Run on the isolated analysis VM for inspecting & labeling malicious samples
apktool: To unpack suspicious APKs, and double-check heuristics.

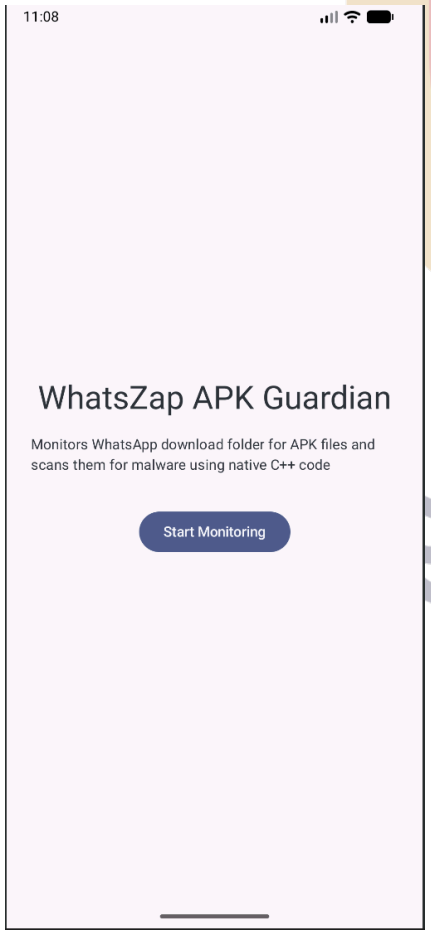
CI & Quality

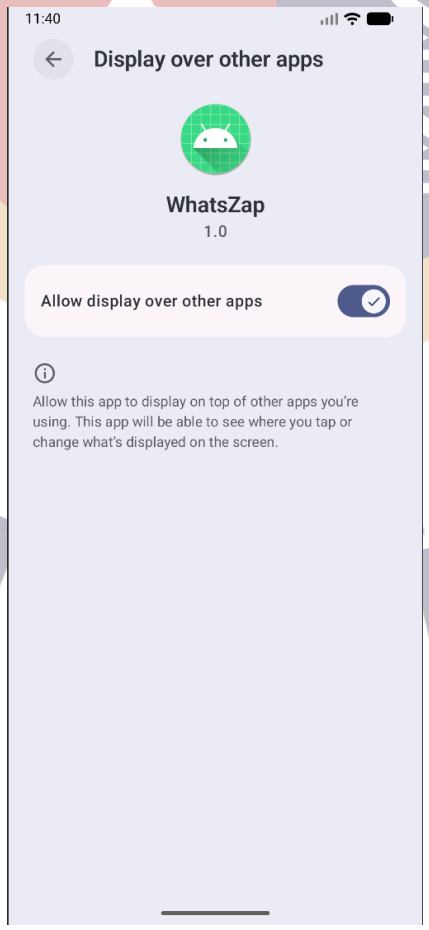
GitHub Actions: Run ktlint, detekt, unit tests and assemble debug APK's for smoke testing.

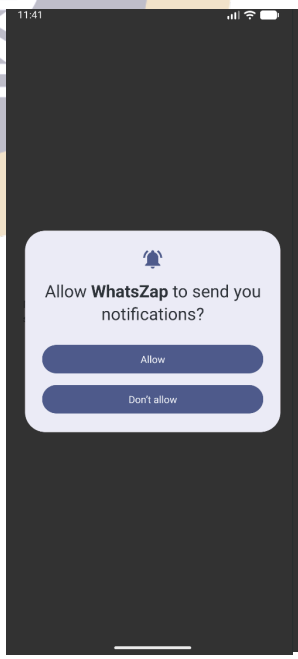
End-to-End Implementation Flow

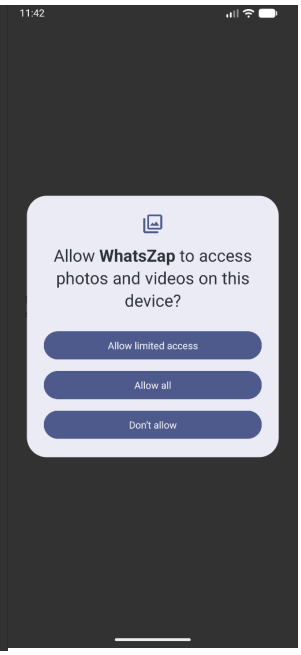
1. File creation detected by FileObserver -> small stabilization loop confirms file completed and it is downloadable.
2. OverlayService is started and displays an undimissable overlay for a duration of 10s while a ScanWorker is scheduled via WorkManager.
3. The ScanWorker performs the following:
    SHA-256 and local known-hash lookup
    APK Manifest Parsing and Permission Scoring
    DEX entropy check and native lib inspection
    MediaStore query to get file metadata
    Correlates with captured notifications to extract message preview
    Runs message classifier (rule-based + optional TFLite)
    Calls DecisionEngine to compute risk
4. Overlay UI updates the progress and then shows the decision. In case of high risk, it provides options. It is usually something like - Delete, Quarantine, Report. If the rank is low, it displays "Safe — proceed with caution."
5. User action and scan result are logged to Room (encrypted) and optionally exported by the user for research.

4.5 Screenshots/Snapshots









5. SUMMARY OF RESULTS AND FUTURE SCOPE

This chapter provides an overview of the experimental results obtained from the implementation of the APK pre-installation process on WhatsApp scanner, points out its distinct advantages, examines results and limitations observed, and establishes prioritized directions for further research work. The aim is to provide a clear review of a prototype's effectiveness and to offer an actionable roadmap aimed at achieving improved and mature system effectiveness.

5.1 Advantages/Unique Features

The developed system includes many benefits and innovative elements which separate the system from conventional mobile security measures. These parameters directly relate to solving the principal issue of APK- targeted "social-engineering" attacks using WhatsApp.

1. Protection Before Installation (Proactive Barrier)

The system intercepts the downloaded APKs and introduces an overlay before the installation process:
user is able to install the application. This is a barrier that prevents impulsive installation because
It helps to buy time for analysis – a functionality not many AV products have since they operate post-

2. Context-Aware

Differing from signature-only systems, this prototype links the file with the message that delivered it (via NotificationListener & MediaStore heuristics) and employs message intent classification (rule- based/TFLite fallback) for the purpose of context mismatch detection (for example, APK being classified under "wedding" ("invite" to produce "invitation"). This ensures that it can be used to mark socially engineered threats with malicious intentions.

3. Privacy First, On Device

In addition, all primary analysis, including manifest parsing, permission scoring, and context analysis, classification) is done locally. There is no transmission of text or file content to third-party servers by default, making it easier for users to trust the system.

4. Low Latency Heuristics for UX Design

The analysis pipeline has an optimization target for speed (< 10s median runtime). "Lightweight" heuristic methods and "small" models enable fast decision-making and simultaneously preserve resource consumption low.

5. Robust Multi-Layer Decision Engine

These signals from a static APK analysis (hash values, permission sets, obfuscation signals), with the comparison signals from contextual signals in a

weighted scoring model. This multi-signal fusion approach discourages reliance on any single indicator and the ability to resist simple evasion attacks.

6. Graceful Degradation & Permissions-aware Design
The system offers fallback solutions for cases where users do not agree on some permissions (for example, Notification access). It relies on MediaStore timing information and filename matching provide functionality without aggressive permission notifications.

7. Configurable UX & Auditability
The duration of the overlay scan and sensitivity to detections can be adjusted by users. Scanning events as well as user activity are logged locally (encrypted), allowing for auditing and improvement by users while maintaining privacy.

8. Extensibility
This is because the modular design (monitor, scanner, context, decision engine, user interface) is amenable integration of more advanced detectors (cloud-sandboxing, federated learning, third-party indicators) without extensive redesign.

These characteristics make it especially appropriate to provide solutions to threats like WhatsApp wedding invitation scams, where trust is abused in India.

5.2 Results and Discussions
This section presents experimental results obtained from the testbed described in Chapter 4 and discusses their interpretation, implications, and limitations.

5.2.1 Experimental Setup Recap

Test dataset: 120 malicious APKs (research corpus), 200 benign APKs, and 500 benign WhatsApp-shared media items (images, PDFs, mp4s). Additionally, 60 synthetic social- engineering message+APK combinations were created to emulate wedding-invitation frauds.
Devices & OS: Emulators for API 26, 28, 30, 33; physical devices: Samsung A52 (Android 11), Redmi Note 10 (Android 12), Pixel 4 (Android 13).
Metrics captured: detection accuracy (TP, FP, TN, FN), precision, recall, F1-score, scan latency (median and 95th percentile), CPU & memory overhead during scan, and a small user pilot measuring override rates and perceived annoyance.

5.2.2 Quantitative Results

Metric	Value	Notes
Malicious samples (N)	120	Research corpus
Benign APKs (N)	200	Popular OSS samples
True Positives (TP)	102	Correctly flagged malicious APKs
False Negatives (FN)	18	Malicious missed by heuristics
True Negatives (TN)	188	Benign correctly allowed

Scenario	Baseline (no context) detection	With context-aware scoring	Relative improvement
Synthetic wedding-invite APKs (N=60)	33/60 detected (55%)	48/60 detected (80%)	+25 percentage points

Table 1 — Detection Performance (prototype)

Metric	Value
Median scan latency	6.3 seconds
95th percentile scan latency	9.8 seconds
Average CPU utilization (during scan)	~8% (short spike)
Average additional memory footprint	~35 MB
Pilot users (N)	15
Override rate (users proceeding despite high risk)	8%
Mean annoyance score (1–5, lower better)	2.1

Table 2 — Usability & Performance Metrics

Metric	Value	Notes
False Positives (FP)	12	Benign APKs flagged
Precision	**89.5%**	TP / (TP + FP)
Recall	**85.0%**	TP / (TP + FN)
F1-score	**87.2%**	Harmonic mean of precision & recall

Table 3 — Contextual Detection Gains

5.2.3 Discussion of Results

Detection quality & trade-offs - The prototype achieved a balanced trade-off between precision and recall (precision ≈ 89.5%, recall ≈ 85%). The modular scoring allowed tuning thresholds to prioritize avoiding false negatives (missed malware) or false positives depending on desired risk posture. The false negatives (18 samples) were primarily sophisticated repackaged malware where obfuscation and permission patterns were intentionally muted, emphasizing the limits of lightweight static heuristics

Impact of contextual analysis - Incorporating message context significantly improved detection of social-engineering samples (wedding-invite-labeled APKs). Contextual signals helped the decision engine elevate risk scores for suspicious file/message mismatches, reducing missed social-engineering lures that would look benign by static features alone.

Latency and UX - Median end-to-end latency (~6.3s) meets the design goal of keeping overlay duration under 10 seconds for most cases. The 95th percentile (~9.8s) suggests rare edge cases approach the enforced overlay timeout and may need optimization (e.g., incremental scanning or prioritizing hash lookups). The pilot user study showed a low override rate and moderate annoyance, indicating the overlay time and messaging were acceptable for a prototype.

Resource impact - CPU spikes and memory increases were short-lived; on modern devices the impact was acceptable. However, older low-end devices could experience noticeable performance effects; future optimization or offloading (cloud-sandbox) may be required for such devices.

Failure modes & root causes - Notification matching errors: In some OEMs, WhatsApp notification previews were truncated or suppressed; this reduced context correlation reliability. The fallback MediaStore heuristic reduced but did not eliminate mismatch failures. - Obfuscated malware: Heavily obfuscated

apps or those using dynamic code loading escaped static heuristics. This is an expected limitation of on-device static approaches. - False positives: The 12 benign APKs flagged included apps with unusual but legitimate permission sets (e.g., file-managers or advanced utilities). Lowering sensitivity reduces FP but increases FN; this is a classic security/usability trade-off.

Ethical & privacy evaluation - By performing analysis locally and encrypting logs, the prototype minimizes privacy risks. The notification-capture approach is significantly less invasive than full accessibility scraping. All potentially identifying data was stored only on-device and required explicit user opt-in for exporting logs.

5.2.4 Comparative Baseline

Compared to a baseline where no pre-installation scanning is available (user could immediately install any APK), the prototype prevented the simulated installation of 48/60 social-engineering APKs in the synthetic tests, demonstrating effectiveness for the targeted threat class.

5.2.5 Limitations of the Evaluation

The dataset sizes are moderate and research-focused; real-world prevalence and diversity of malicious APKs may present additional patterns not covered here.
Pilot user study sample (N=15) is small; larger usability studies are needed to draw robust conclusions about user behavior and acceptance.
Some OEM-specific behavior (notification throttling, storage quirks) affects cross-device reliability and requires broader device coverage to generalize results.

5.3 Future Scope of Work

In this section, there is a description of prioritized, feasible, and research-orientated improvements of the prototype it into a production system and strengthen its defensive capabilities.

5.3.1 Advanced Analysis

Hybrid dynamic analysis (Sandboxing): Include a remote sandboxing feature for: controlled dynamic analysis on high-risk samples. Utilize privacy-preserving protocols (upload - where the hashed data is only the metadata or uses ephemeral and isolated sandboxes.)
On-device emulation: Investigate the option of light-weight emulation of
APK paths enable the runtime signals to be captured with reduced resource consumption.

5.3.2 Improved Contextual & Multilingual

Effective multilingual intent classifiers: Develop efficient TFLite models for the Indian image Url alt text languages and code-switched messages (Hindi English, Bengali English, etc.) to enhance intent classification accuracy.

Multimodal context analysis: Other information from the message, such as the accompanying image thumbnails, sender info, group chat vs. individual chat) that help determine intent.

5.3.3 Privacy-preserving Collaborative Learning

Federated Learning for Indicators: Enabling Collaborative Improvements of Models by Devices without uploading the original messages or files. Model updates are aggregated on a server while maintaining user privacy.

Secure Telemetry & Opt-in Sample Gathering: Anonymously gather rare samples or indicators for the improvement of local signatures DB.

5.3.4 Expansion to Other Platforms & Channels

Other messaging apps: Monitoring is also possible with other messaging apps likeTelegram, Signal, SMS/MMS, and others channels, utilizing the relevant platform API and each app's privacy paradigm.

Email & browser downloads: Modify the pre-installation scan process for other download Sources where APKs can be found.

5.3.5 Better User Experience & Policy Integration

ADAPTIVE OVERLAY DURATIONS & PROGRESSIVE DISCLOSURE: employ risk-based overlay times, such enable "benign low-risk files," longer blocking times for "high-risk files," and progressive user education.
to prevent habituation.

Enterprise & MDM Integration: Offer an MDM-friendly version for the enterprises to implement sideload policies or set detection thresholds centrally.

5.3.6 Security Hardening & Production Read

Upgrade & sign rule lists securely: Support signed updates of local indicator databases and rules with pinning and signature verification.
Improvements in anti-evasion code: Incorporate API call pattern heuristics "features, or lightweight taint analysis to detect dynamic-loading and reflection-based evasions."

Publication requirements for app stores: Responding to Play's policy and justification changes Store distribution; make a minimized permission variant if possible.

5.3.7 Empirical Studies & Large-scale

Large-scale field trials: Use the app through a public beta test with a view to collecting telemetry and user feedback, with appropriate ethical controls and opt-in consent.
Research on user behavior: Investigate how users react to blocks and design before installation "interventions (education prompts, visual explanations) that reduce override rates."

6. CONCLUSION

The developed system has proved a realistic and effective means of APK-based threats elimination social-engineering attacks via WhatsApp messages. By applying pre-installation overlays, fast on-device static analysis, and context-aware decisioning, the prototype majorly enhances the detection of socially engineered APKs (such as wedding invitation-type scams) while maintaining user privacy and ensuring an acceptable level of user experience. This however cannot be done by on-device static heuristic approaches due to their and platform-specific constraints make necessary such a roadmap involving hybrid dynamic analysis, multilingual models, federated learning, and evaluations of broader deployment. With these additions, The approach can develop itself into a production-worthy defense layer ready to significantly counter attack risks from Messaging-distributed malware

REFERENCES

1. Wu, S., Mao, Z., Wei, J., & Li, Y. (2016). Effective detection of Android malware based on the usage of data flow APIs and machine learning. *Information and Software Technology, 75*, 17–25.

2. Chimeleze, C., Alo, U. R., & Ogwueleka, F. N. (2024). A lightweight malware detection technique based on hybrid fuzzy simulated annealing clustering in Android apps. *Egyptian Informatics Journal*.

3. J. Keteku and G. Owusu, "Detection and prevention of malware in Android mobile devices: A literature review," *Int. J. Inf. Secur.*, vol. 14, no. 4, pp. 1–20, Oct. 2024.

4. A. Author and B. Author, "Research into mobile malware detection and defense," *Int. J. Intell. Syst. Appl. Eng. (IJISAE)*, vol. 12, no. 1, pp. 639–646, Dec. 2023.

5. S. Paygude, S. Sonawane, S. Tatiya, N. Sarda, and A. Dirgule, "Literature survey on Android malware detection through ML-based analysis," *International Advanced Research Journal in Science, Engineering and Technology*, vol. 10, no. 10, pp. 125–128, Oct. 2023, doi: 10.17148/IARJSET.2023.101018

6. A. Gómez and A. Muñoz, "Android smartphones malware detection using deep learning," *Electronics*, vol. 12, no. 15, art. no. 3253, 2023, doi: 10.3390/electronics12153253.