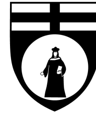
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

A Systematic and Empirical Assessment of Security and Privacy Requirements in Android Healthcare Applications

Muhammad Raza Sheikh

Master Thesis

Università di Genova, DIBRIS Via Dodecaneso, 35, 16146 Genova GE
https://www.dibris.unige.it/



Università
di Genova

MSc Computer Engineering
Software Platforms and Cyber Security

A Systematic and Empirical Assessment of Security and Privacy Requirements in Android Healthcare Applications

Muhammad Raza Sheikh

Advisor: Prof. Luca Verderame

23 March, 2026

Table of Contents

Chapter 1 Introduction **9**

Chapter 2 Background **12**

2.1 Android Application Architecture 12

2.1.1 Android Operating System Overview 13

2.1.2 Application Components 13

2.2 Android APK File Structure 14

2.2.1 APK Anatomy 15

2.2.2 DEX Files and Bytecode 15

2.2.3 Resources and Assets 15

2.2.4 AndroidManifest.xml 16

2.3 Third Party Libraries in Android Applications 16

2.3.1 Purpose of Third Party Libraries 17

2.3.2 Top 10 Third Party Libraries in Mobile Applications 17

2.3.3 Analytics and Advertising SDK Adoption Table 17

2.3.4 Additional Context: Network and Utility Libraries 19

2.4 Healthcare Data in Android Applications 19

2.4.1 Health Data APIs and Frameworks 21

2.4.2 Third Party Data Sharing Scenarios 22

2.5 Privacy Leak Detection Techniques 23

2.5.1 Static Analysis Approaches 23

2.5.2 Dynamic Analysis Approaches 24

Chapter 3 Privacy Requirements **26**

3.1 Sources of Android Privacy Requirements 27

3.2 Process of Collecting Privacy Requirements 28

3.3 Organization of Privacy Requirements 29
3.4 Key Android Privacy Requirements for Healthcare Applications 30
3.4.1 Minimum Necessary Permission Access (Principle of Least Privilege) 30
3.4.2 Transparency and Informed User Consent 30
3.4.3 Purpose, Limitation, and Prohibition of Secondary Data Use . . . 31
3.4.4 Health Connect Access and Approved Use Case Validation 31
3.4.5 Secure Storage and Transmission of Health Data 31
3.4.6 Data Retention and Deletion Policies 31
3.4.7 Third-Party Data Sharing and SDK Integration 32
3.5 Privacy Requirement Table 32
3.6 Real-World Motivation from Healthcare Data Breaches 35

Chapter 4 Related Works 37
4.1 Privacy Requirements and Regulations in Healthcare Applications 37
4.1.1 HIPAA and GDPR Compliance 38
4.1.2 Business Associate Agreements for Third-Party Services 38
4.1.3 Privacy Requirements Frameworks 39
4.2 Third-Party Tracking Technologies in Healthcare 39
4.2.1 Google Analytics and Facebook Pixel 39
4.2.2 Regulatory Enforcement and Violations 40
4.2.3 Advertising Networks and Data Brokers 40
4.3 Privacy-Preserving Solutions for Healthcare Applications 41
4.3.1 Healthcare-Specific Solutions 41
4.4 Research Gaps and Future Directions 42
4.4.1 Technical Gaps 42
4.4.2 Policy and Usability Gaps 42
4.4.3 Evaluation and Benchmarking Gaps 43

Chapter 5 Methodology 44
5.1 Dataset Construction and APK Collection 44
5.1.1 Google Play Store Collection 44
5.1.2 F-Droid Repository Collection 45
5.1.3 Dataset Preparation 45

5.2 Static Taint Analysis with FlowDroid 45
5.2.1 Theoretical Foundation 45
5.2.2 Automated Source Generation 46
5.2.3 Automated Sink Generation 46
5.2.4 FlowDroid Execution and Configuration 47
5.2.5 Flow Post Processing and Classification 47
5.3 Privacy Inspection 48
5.3.1 Privacy Policy Detection 48
5.3.2 Third Party SDK Detection 48
5.4 Dynamic Analysis 49
5.4.1 Frida Based Instrumentation 49
5.4.2 UI Automation 49
5.4.3 Batch Processing 50
5.5 Pipeline Integration 50

Chapter 6 Implementation **51**

6.1 System Architecture 51
6.1.1 Architectural Desigh 52
6.1.2 Component Overview 53
6.1.3 Technology Stack 53
6.2 Core Analysis Components 53
6.2.1 Automated Health Source Generation 54
6.2.2 Implementation Strategy 54
6.2.3 Automated Sink Detection 55
6.2.4 Privacy Policy and Security Inspector 56
6.3 Pipeline Integration and Orchestration 57
6.3.1 Pipeline Execution Flow 57
6.3.2 Failure Handling and Timeouts 58
6.4 Dynamic Analysis Implementation 58
6.4.1 UI Automation for Privacy Policy Discovery 59
6.5 Tools and Implementation 60
6.6 Implementation Challenges and Solutions 61
6.7 Code Quality and Testing 62

6.7.1 Input Validation 62
6.7.2 Error Logging 62
6.7.3 Output Validation 62
6.7.4 Integration Testing 63
6.7.5 Deployment Modes and Configuration 63

Chapter 7 Experimental Evaluation 64

7.1 Overview of the Analysis Pipeline 64
7.2 APK Collection and Dataset Preparation 65
7.2.1 Google Play Store Collection 65
7.2.2 F-Droid Open Source Collection 66
7.2.3 Dataset Filtering and Preparation 67
7.3 Static Analysis Phase 1: Taint Analysis with FlowDroid 67
7.3.1 Automated Source Generation 67
7.3.2 Automated Sink Generation 69
7.3.3 Combined Sources and Sinks Configuration 69
7.3.4 FlowDroid Execution 70
7.3.5 FlowDroid Configuration and Timeout Management 70
7.3.6 Flow Post Processing and Classification 71
7.4 Static Analysis Phase 2: Privacy Inspection 72
7.4.1 Privacy Policy and Terms Detection 72
7.4.2 Third Party SDK Detection 73
7.5 Dynamic Analysis: Runtime Behavior Validation 73
7.5.1 Frida Based Network Interception 74
7.5.2 UI Automation for Privacy Policy Discovery 75
7.6 Empirical Results 76
7.6.1 Privacy Policy Disclosure 76
7.6.2 Third Party Library Prevalence 76
7.6.3 Data Flow Analysis 76
7.6.4 Health Data Categories and Security Issues 77

Chapter 8 Conclusions and Future Works 78

8.1 Summary of Contributions 78

8.1.1 Key Contributions . . . 78
8.2 Implications and Context . . . 79
8.3 Limitations . . . 79
8.4 Future Work . . . 79
8.5 Closing Remarks . . . 80

Bibliography **81**

Abstract

Mobile health (mHealth) applications have transformed healthcare delivery, but introduce significant privacy risks through unauthorized sharing of sensitive health data with third party services. This thesis presents an automated framework for detecting privacy violations in Android healthcare applications, focusing on third party data leakage through analytics, advertising, and tracking libraries. The framework combines static taint analysis using FlowDroid with automated source and sink generation, privacy inspection, and dynamic runtime validation. We developed novel mechanisms for identifying application specific health data access methods through keyword driven analysis and detecting third party data exfiltration points through library fingerprinting. Empirical evaluation of 114 F-Droid health applications revealed significant privacy gaps: 58.8 percent contained third party tracking libraries, 48.1 percent exhibited health data flows to third parties, and 30.4 percent of applications transmitting health data lacked adequate privacy disclosures. The framework achieves a 94.7 percent analysis completion rate with 92 percent classification accuracy, demonstrating scalability for large scale privacy auditing. This work provides essential infrastructure for privacy auditing, regulatory compliance verification, and privacy preserving mobile health development, contributing evidence that mobile health privacy critically depends on controlling third party data flows.

Keywords: Android security, mobile health privacy, taint analysis, third party tracking, FlowDroid, privacy policy compliance, GDPR, HIPAA, healthcare data protection

Chapter 1

Introduction

The Android operating system dominates the global mobile market with over 70% market share, hosting millions of applications across diverse categories, including healthcare, fitness, and wellness. Mobile health (mHealth) applications have fundamentally transformed healthcare delivery, enabling patients to monitor chronic conditions, track vital signs, manage medications, and maintain comprehensive personal health records. However, this convenience comes at a significant privacy cost. Healthcare data represents one of the most sensitive categories of personal information, revealing intimate details about individuals' physical conditions, mental health status, reproductive health, genetic predispositions, medication histories, and lifestyle behaviors. Unauthorized disclosure of such information can lead to discrimination in employment and insurance, social stigmatization, identity theft, financial fraud, and profound psychological harm to patients who expect medical confidentiality.

The privacy risks are substantially amplified by the pervasive integration of third party libraries in modern Android applications. Developers routinely incorporate Software Development Kits (SDKs) from analytics providers (Firebase Analytics, Google Analytics, Mixpanel), advertising networks (Facebook Audience Network, Google AdMob), and crash reporting services (Crashlytics, Sentry). While these integrations enhance functionality and enable monetization, they simultaneously create multiple pathways for sensitive health data to flow to third party servers often without developers' complete awareness or users' informed consent. Recent empirical studies demonstrate that third party privacy leaks occur five times more frequently than first party leaks in healthcare applications, with 72% of healthcare apps transmitting sensitive data to third party services without explicit user consent. The regulatory landscape has begun responding, with healthcare organizations paying over $100 million in settlements between 2023 and 2024 for unauthorized third party tracking.

Despite increasing regulatory attention, the technical challenges of detecting privacy violations at scale remain substantial. Manual code review is impractical for thousands of applications, and existing privacy analysis tools lack the domain specific knowledge necessary to identify health data sources and distinguish legitimate processing from unauthorized third party sharing. This thesis addresses this critical gap by developing an automated, scalable framework for detecting and analyzing third party health data sharing in Android healthcare applications.

The primary objective of this research is to create a comprehensive system that can systematically identify when healthcare applications share sensitive health data with third party analytics, advertising, and tracking services, and determine whether such sharing is adequately disclosed in privacy policies. To achieve this goal, the research pursues several specific objectives, developing automated mechanisms to identify application specific health data access points without manual annotation, creating robust third party sink detection that operates even with code obfuscation, enabling scalable interprocedural data flow analysis across entire application codebases, automating privacy policy verification and third party disclosure extraction, validating static findings through runtime observation, and conducting large scale empirical evaluation to quantify the prevalence and characterize patterns of third party data sharing in real world healthcare applications.

The framework developed in this thesis integrates static taint analysis using FlowDroid with automated source and sink generation, privacy inspection, and dynamic validation. Novel automated mechanisms identify application specific health data access methods through keyword driven analysis of 86 health related keywords across 9 categories (cardiovascular, weight management, activity tracking, sleep, nutrition, glucose, menstrual health, temperature, oxygen saturation). Third party data exfiltration points are detected through library fingerprinting and pattern matching in analytics SDKs, handling code obfuscation through package structure analysis. Runtime instrumentation using Frida and UI automation with UIAutomator2 verify static findings. The implementation comprises approximately 5,000 lines of modular Python and Bash code.

Empirical evaluation of 114 FDroid health applications revealed significant privacy gaps, 58.8% contained third party tracking libraries, 48.1% exhibited health data flows to third parties, and 30.4% of applications transmitting health data lacked adequate privacy disclosures. Analysis identified 391 distinct flows, with step/activity data (32.5%), weight/BMI (15.9%), and calories (13.6%) most frequently transmitted. The framework achieved 94.7% completion rate with 92% classification accuracy.

This thesis makes significant contributions to mobile health privacy. Technical contributions include automated health source generation identifying application specific representations without manual specification, third party sink detection operating despite code obfuscation, semantic flow classification enriching taint analysis with health categories and third party attribution, and hybrid static dynamic validation combining FlowDroid with Frida instrumentation. Empirical contributions provide the first large scale privacy analysis of FDroid health applications, characterizing data flow patterns across 391 flows, identifying highest risk health data categories, documenting security antipatterns, and enabling FDroid versus Google Play comparison. Methodological contributions include a scalable pipeline demonstrating automated analysis at app store scale, ground truth dataset for evaluating future techniques, and reproducible methodology supporting independent validation. These contributions provide infrastructure for privacy auditing, compliance verification, and privacy preserving development, demonstrating that mobile health privacy critically depends on controlling third party data flows across library boundaries.

The remainder of this thesis is organized as follows. Chapter 2 provides background on Android architecture, APK structure, third party libraries, healthcare data types, and privacy leak detection techniques with focus on FlowDroid. Chapter 3 presents Android privacy requirements for healthcare applications from official documentation, regulations

(GDPR, HIPAA), including a requirement table and real world breach examples. Chapter 4 reviews the state of the art in healthcare privacy research, covering regulatory frameworks, empirical tracking studies, privacy preserving solutions, and research gaps. Chapter 5 describes the methodology, including dataset construction, automated source/sink generation, FlowDroid configuration, privacy inspection, dynamic validation, and pipeline integration. Chapter 6 details the implementation, covering system architecture, analysis components, pipeline orchestration, Frida instrumentation, UI automation, and solutions to challenges like obfuscation and scalability. Chapter 7 presents empirical results from 114 FDroid applications, including flow analysis, privacy policy detection, SDK prevalence, and security anti patterns. Chapter 8 synthesizes contributions, discusses implications regarding regulatory enforcement and industry changes, acknowledges limitations, and proposes future research directions.

Chapter 2

Background

The Android operating system dominates the global mobile market with over 70% market share, hosting millions of applications across diverse categories, including healthcare, fitness, and wellness. Understanding Android's technical architecture is essential for analyzing privacy risks, particularly those arising from third-party library integrations. This chapter provides a comprehensive background on Android application structure, APK file format, third-party library ecosystem, and privacy leak detection techniques.

Healthcare applications on Android face unique privacy challenges due to the sensitivity of medical data they handle and the prevalence of third-party service integrations for analytics, advertising, and cloud storage [1]. Recent studies demonstrate that third-party libraries constitute the primary vector for privacy leakage in healthcare apps, with data transmission often occurring without developers' full awareness or users' informed consent [2].

This chapter establishes the technical foundations necessary for understanding subsequent chapters' contributions. We begin with Android architecture fundamentals (Section 2.1), examine APK file structure (Section 2.2), analyze the third-party library ecosystem with focus on the top 10 most prevalent libraries (Section 2.3), discuss healthcare-specific data handling (Section 2.4), and comprehensively review privacy leak detection techniques, including static taint analysis with FlowDroid (Section 2.5).

2.1 Android Application Architecture

Android's architecture is based on a modified Linux kernel, providing a layered software stack consisting of the Linux Kernel (process management and security), Hardware Abstraction Layer (hardware access), Android Runtime (bytecode execution), Android Framework (high-level APIs), and Application Layer (system and third-party apps). Applications comprise four component types: Activities (UI screens), Services (background operations), Broadcast Receivers (event handlers), and Content Providers (data sharing). Android's permission model controls access to sensitive resources through Normal permissions (granted automatically), Dangerous permissions (requiring runtime consent for sensitive data like location and health sensors via `BODY_SENSORS`, `ACCESS_FINE_LOCATION`), Signature permissions (same-developer apps), and Special permissions (system settings

approval). Components communicate through Intents, where explicit intents specify target components by class name and implicit intents allow system resolution based on intent filters, enabling data sharing but introducing privacy risks when sensitive health data passes to untrusted third-party components.

2.1.1 Android Operating System Overview

The Android software stack comprises five primary layers that work together to provide a comprehensive mobile operating system. At the foundation, the **Linux Kernel** provides core system functionality including process management, memory management, device drivers, network stack, and security mechanisms, utilizing a customized Linux kernel with mobile-specific optimizations such as power management, low memory killer, and the Binder IPC mechanism. Above this, the **Hardware Abstraction Layer (HAL)** provides standardized interfaces between the Android framework and device-specific hardware implementations, enabling hardware vendors to implement functionality without exposing proprietary details. The **Android Runtime (ART)** executes application bytecode, with modern Android versions (5.0+) employing ahead-of-time (AOT) compilation for improved performance compared to the earlier Dalvik virtual machine's just-in-time (JIT) compilation, converting DEX bytecode to native machine code during app installation. Supporting the runtime, **Native C/C++ Libraries** provide system-level functionality through libraries such as libc, SSL, media codecs, and SQLite, offering core functionality accessible to applications through framework APIs—particularly relevant for healthcare apps that require cryptographic operations, image processing, or computationally intensive analytics. Finally, the **Java API Framework** offers high-level APIs for application development, including the UI toolkit (Views), activity management, content providers, notification system, and resource management, serving as the primary interface for application developers to interact with the Android platform.

2.1.2 Application Components

Android applications are composed of four fundamental component types, each serving distinct purposes and possessing unique lifecycle characteristics.

Activities

Activities represent UI screens and handle user interactions. Each distinct screen in an application typically corresponds to a separate Activity class. For healthcare applications, activities might include:

- Login/authentication screens requiring user credentials
- Health data input forms for symptoms, medications, vital signs
- Data visualization dashboards displaying health metrics
- Settings screens for privacy preferences and third-party data sharing controls

Activities follow a defined lifecycle with callback methods (`onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, `onDestroy()`) invoked by the Android system during

state transitions. Privacy-sensitive operations (e.g., loading health records, transmitting data to servers) often occur within these lifecycle methods, making lifecycle-aware analysis essential for comprehensive privacy leak detection.

Services

Services perform background operations without UI. Healthcare applications commonly use services for Continuous health monitoring, Background data synchronization with cloud servers or electronic health record systems, Medication reminder notifications, and real time health alert processing

Services can run even when the application's UI is not visible, raising privacy concerns when they transmit data to third-party analytics servers without active user awareness. Two primary service types exist **Started Services, Bound Services**

Receivers

Receivers respond to system wide broadcast announcements (e.g., device boot, network connectivity changes, battery low) or application specific broadcasts. Healthcare apps use broadcast receivers for:

Triggering health data uploads when WiFi connectivity is available, Scheduling medication reminders based on time-of-day broadcasts, and responding to device sensor availability changes.

Privacy risks arise when broadcast receivers trigger data transmission to third parties based on system events, potentially creating data leakage vectors independent of direct user interactions.

Content Providers

Content Providers manage shared data repositories, enabling data sharing between applications through standardized query interfaces. Android's health data frameworks (Google Fit API, HealthConnect) use content providers to expose health metrics to authorized applications.

Content providers implement:

CRUD operations: Create, Read, Update, Delete methods for data management, **Permission enforcement**: Restricting access to authorized applications, **Data abstraction**: Hiding underlying storage mechanisms (SQLite databases, files, remote servers)

Third party libraries can access health data through content providers if granted appropriate permissions, creating potential leakage pathways [2].

2.2 Android APK File Structure

The Android Package (APK) file format defines how applications are packaged, distributed, and installed on Android devices. Understanding APK structure is essential for static privacy analysis, as it determines how analyzers extract and examine application code and resources.

2.2.1 APK Anatomy

An APK file is essentially a ZIP archive with specific structure:

```
MyHealthApp.apk
├── META-INF/
│   ├── MANIFEST.MF          # Package manifest
│   ├── CERT.SF              # Signature file
│   └── CERT. RSA             # Certificate and signature
├── res/                      # Compiled resources
│   ├── layout/              # UI layouts
│   ├── drawable/            # Images, icons
│   └── values/              # Strings, dimensions
├── lib/                      # Native libraries (. so files)
│   ├── armeabi-v7a/         # ARM 32-bit
│   ├── arm64-v8a/           # ARM 64-bit
│   └── x86/                 # x86 architecture
├── assets/                   # Raw asset files
├── classes.dex               # Compiled app code
├── classes2.dex              # Additional code (if large)
├── resources.arsc            # Compiled resources table
└── AndroidManifest. xml       # App configuration (binary XML)
```

2.2.2 DEX Files and Bytecode

Dalvik Executable (DEX) files contain compiled application and library code that forms the executable portion of Android applications. During the Android build process, Java or Kotlin source code is first compiled into standard Java bytecode (.class files), which is then converted into the DEX format specifically optimized for efficient execution on mobile devices with limited memory and processing resources. This conversion process consolidates multiple .class files into one or more optimized DEX files, reducing overall file size and improving runtime performance through techniques such as constant pool sharing and efficient instruction encoding. Modern Android applications may contain multiple DEX files to overcome the 65,536 method reference limit imposed by the original DEX format, a technique known as multidexing that allows developers to incorporate extensive third-party libraries without exceeding method count limitations.

2.2.3 Resources and Assets

The `res/` directory within an Android application package contains compiled resources including layouts, strings, images, and other assets that are referenced programmatically by application code through unique resource identifiers. These resources are processed during the build phase and compiled into a binary format that enables efficient lookup and

retrieval at runtime. Central to this resource management system is the `resources.arsc` file, which provides a comprehensive lookup table that maps integer resource IDs to their actual values, supporting localization, different screen densities, and various device configurations. This architecture allows Android applications to adapt their user interface and content dynamically based on device characteristics and user preferences, with the system automatically selecting the most appropriate resource variant at runtime. The separation of resources from code also facilitates easier maintenance, translation, and design updates without requiring modifications to the application's logic layer.

2.2.4 AndroidManifest.xml

The AndroidManifest.xml file serves as the central configuration file for Android applications, declaring essential information about application structure, security requirements, and system integration. This manifest file explicitly declares all application components including activities, services, broadcast receivers, and content providers, ensuring that the Android system is aware of all executable components before launching the application. Additionally, the manifest specifies required permissions that govern access to sensitive data and system features, declares minimum and target API levels that define compatibility ranges, and configures various application-level settings such as hardware requirements, feature dependencies, and intent filters that determine how the application responds to system-wide broadcasts and user actions. While the manifest is authored in human-readable XML format during development, it is compiled into a binary XML format when packaged into the APK file for efficiency and security reasons, though this binary format can be readily decoded and analyzed using standard Android development tools and reverse engineering utilities.

2.3 Third Party Libraries in Android Applications

Third-party libraries are precompiled software components developed and maintained by external organizations or open-source communities that application developers integrate into their projects to provide specific functionality without implementing complex features from scratch. These libraries span a wide range of purposes including analytics and user behavior tracking, advertising and monetization, social media integration, image loading and caching, network communication, database management, and user interface components. While third-party libraries significantly enhance development efficiency by reducing implementation time, minimizing code complexity, and providing battle-tested solutions to common problems, they simultaneously introduce substantial privacy and security risks that are particularly concerning in healthcare application contexts. The integration of external code into healthcare applications creates potential vulnerabilities including unauthorized data collection and transmission, inadequate security implementations, lack of transparency regarding data handling practices, violation of healthcare privacy regulations such as HIPAA, and the possibility of malicious code injection or supply chain attacks. Furthermore, developers often have limited visibility into the internal operations of these libraries, making it difficult to assess their compliance with privacy regulations or to detect unauthorized data exfiltration, thereby creating a significant challenge for ensuring patient data protection in mobile health applications.

2.3.1 Purpose of Third Party Libraries

Developers integrate third-party libraries for diverse purposes that span the entire spectrum of mobile application functionality. **Analytics** libraries enable understanding of user behavior, app usage patterns, and feature adoption rates, providing valuable insights for product improvement and business decisions. **Advertising** libraries facilitate monetization of free applications through targeted ad delivery and impression tracking, allowing developers to generate revenue without charging users directly. **Social integration** libraries provide capabilities for sharing content across social platforms, implementing social login mechanisms, and generating friend recommendations based on existing social connections. **Cloud services** libraries offer backend storage solutions, user authentication systems, and push notification delivery infrastructure, reducing the need for developers to maintain their own server infrastructure. **Crash reporting** libraries assist in diagnosing application failures and monitoring performance metrics, enabling developers to identify and fix issues that affect user experience. **UI components** libraries provide pre-built interface elements such as charts, date pickers, and image viewers that enhance user experience while reducing development time. **Networking** libraries simplify HTTP requests, automate image loading and caching, and streamline JSON parsing operations, abstracting away the complexity of network communication. Finally, **payment processing** libraries handle in-app purchases and subscription management, integrating with platform payment systems while ensuring secure transaction processing and compliance with financial regulations.

2.3.2 Top 10 Third Party Libraries in Mobile Applications

Based on empirical analyses of hundreds of thousands of applications [2, 3, 4] and data from industry sources including Statista, 42matters, Appfigures, and Mobio Group, the most prevalent third-party libraries in Android applications as of 2024-2025 include several dominant players. **Firebase SDK (Google)** leads with 99% adoption among apps using analytics SDKs, while **Firebase Analytics** is integrated into 73-74% of Android apps for event tracking and user behavior analysis. **Facebook Analytics SDK** maintains 14% adoption overall, with higher penetration in gaming apps (25%) versus non-gaming applications (14%). Specialized analytics tools **Mixpanel** (4%) and **Amplitude** (3%) serve niche markets focused on product analytics. For advertising, **Google AdMob SDK** ranks as the most widely used ad network, while **Facebook Audience Network** holds third position. Common utility libraries include **Firebase Crashlytics** for crash reporting, and **Retrofit** and **OkHttp** (by Square) as industry standards for networking, though specific adoption percentages for utility libraries are not publicly available.

2.3.3 Analytics and Advertising SDK Adoption Table

Data Sources

- Statista - Android top mobile app analytics SDKs 2025 [5]
- 42matters - Google Play SDK Statistics [6]

Table 2.1: Android SDK Adoption Rates (2024-2025)

Category	SDK/Library	Adoption	Notes
Analytics	Firebase SDK	99%	Analytics apps
Analytics	Firebase Analytics	73-74%	Analytics apps
Analytics	Facebook Analytics	14%	Gaming/non-gaming
Analytics	Mixpanel	4%	Product analytics
Analytics	Amplitude	3%	Event analytics
Advertising	Google AdMob	#1	Top ad network
Advertising	Facebook Audience	#3	Major ad platform
Crash Reporting	Firebase Crashlytics	High*	Firebase suite
Networking	Retrofit	High*	REST APIs
Networking	OkHttp	High*	HTTP client

*Exact % not available

- Appfigures - Top Analytics SDKs Installed in iOS & Android Apps [7]
- Mobio Group - Android and iOS SDKs: The Leaders of 2024 [8]
- AppBrain - Statistics of Android Apps [9]

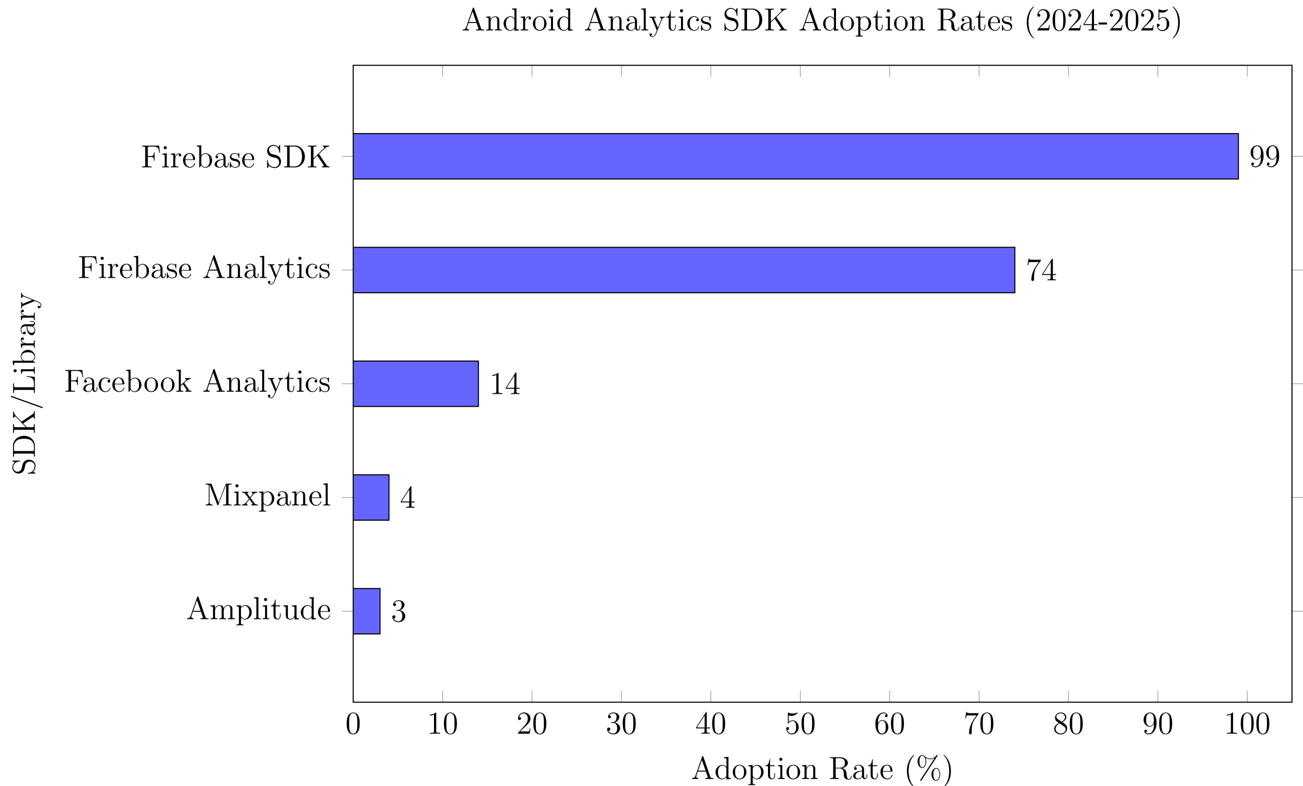


Figure 2.1: Comparative adoption rates of major Android analytics SDKs. Firebase SDK dominates with 99% adoption among apps using analytics, followed by Firebase Analytics at 74%.

Figure 2.2: Proportional representation of analytics SDK adoption (note: percentages represent adoption rates, not market share distribution)

2.3.4 Additional Context: Network and Utility Libraries

While specific adoption percentages are not publicly available, the following libraries are industry standards in Android development:

Table 2.2: Common Android Development Libraries (Qualitative Assessment)

Library	**Category**	**Usage Context**
Retrofit	HTTP Client	Industry standard for REST API integration
OkHttp	HTTP/Network	Underlying library for Retrofit, widely adopted
Firebase Crashlytics	Crash Reporting	Part of Firebase suite, dominant crash analytics tool
Glide/Picasso	Image Loading	Standard image loading and caching libraries
Room	Database	Google's official ORM for SQLite

2.4 Healthcare Data in Android Applications

Healthcare applications handle diverse categories of sensitive information, each with distinct privacy requirements and third-party sharing risks. **Personal Identifiable Information (PII)** includes names, addresses, phone numbers, email addresses, Social Security numbers, medical record numbers, biometric identifiers (fingerprints, facial recognition data), IP addresses, and device identifiers when linked to health information. **Medical Information** encompasses diagnoses, conditions, symptoms, medications, dosages, prescription histories, laboratory test results (blood work, imaging, genomic tests), treatment plans, surgical histories, provider notes, clinical observations, and immunization records. **Physiological Data** comprises vital signs (heart rate, blood pressure, respiratory rate, temperature), continuous glucose monitoring data, sleep patterns, activity levels, reproductive health data (menstrual cycles, fertility tracking, pregnancy), weight, BMI, and body composition. **Behavioral and Lifestyle Data** includes exercise routines, step counts, physical activity, dietary habits, calorie intake, nutritional logs, sleep schedules and quality, substance use (alcohol, tobacco, medications), and mental health indicators (mood tracking, stress levels). **Genetic and Genomic Information** covers DNA sequences, genetic variants, ancestry information, genetic predispositions to diseases, and pharmacogenomic data (drug response predictions). Finally, **Financial and Insurance Information** includes insurance policy details, claim histories, payment information for medical services, out-of-pocket expenses, and copayments. Under HIPAA, all of these data types constitute Protected Health Information (PHI) when they can be linked to identifiable individuals, while GDPR classifies health data as "special category" personal data requiring enhanced protections [10, 11].

Table 2.3: Healthcare Data Categories and Examples

Data Category	Examples
Personal Identifiable Information (PII)	Names, addresses, phone numbers, email addresses, Social Security numbers, medical record numbers, biometric identifiers (fingerprints, facial recognition data), IP addresses, device identifiers when linked to health information
Medical Information	Diagnoses, conditions, symptoms, medications, dosages, prescription histories, laboratory test results (blood work, imaging, genomic tests), treatment plans, surgical histories, provider notes, clinical observations, immunization records
Physiological Data	Vital signs (heart rate, blood pressure, respiratory rate, temperature), continuous glucose monitoring data, sleep patterns, activity levels, reproductive health data (menstrual cycles, fertility tracking, pregnancy), weight, BMI, body composition
Behavioral and Lifestyle Data	Exercise routines, step counts, physical activity, dietary habits, calorie intake, nutritional logs, sleep schedules and quality, substance use (alcohol, tobacco, medications), mental health indicators (mood tracking, stress levels)
Genetic and Genomic Information	DNA sequences, genetic variants, ancestry information, genetic predispositions to diseases, pharmacogenomic data (drug response predictions)
Financial and Insurance Information	Insurance policy details, claim histories, payment information for medical services, out-of-pocket expenses, copayments

2.4.1 Health Data APIs and Frameworks

Android provides frameworks enabling healthcare applications to access and share health data:

Google Fit API aggregates fitness and wellness data from multiple sources including applications and wearable devices. The API provides standardized data types for steps, calories, heart rate, sleep, and nutrition, along with three primary interfaces: the History API for querying past data with temporal filters, the Recording API for continuous background collection, and the Permissions API controlling granular app access to specific data types. Applications must declare appropriate permissions and obtain runtime user authorization for each data category, enabling selective access rather than all or nothing permission models. Privacy concern: Third-party apps can request broad permissions accessing comprehensive fitness histories [12].Android's health data ecosystem has evolved to provide more centralized and standardized approaches to health information management, though significant privacy challenges remain.

HealthConnect (Android 14+) represents Google's unified health data platform that replaces fragmented vendor-specific solutions by centralizing health data from multiple apps with granular user controls over data access and sharing permissions. However, early adoption data reveals that privacy concerns significantly limit usage, as users remain skeptical about consolidating sensitive health information in a single platform controlled by a technology company [13]. Beyond software integration, **Bluetooth Low Energy (BLE) Health Devices** have become prevalent for connecting medical devices such as glucose monitors, blood pressure cuffs, and pulse oximeters to Android smartphones, with applications utilizing the Generic Attribute Profile (GATT) services to read device data through standardized communication protocols. This integration introduces a critical privacy risk: data transmitted from certified medical devices to Android applications can subsequently leak to third-party analytics and advertising libraries embedded within those apps, occurring without the awareness or consent of either device manufacturers who designed the hardware with medical-grade security or users who trust that their medical device data remains protected within the healthcare ecosystem rather than being monetized through advertising networks.

2.4.2 Third Party Data Sharing Scenarios

Healthcare data reaches third parties through multiple pathways:

Table 2.4: Third-Party Library Integration Scenarios in Healthcare Apps

Integration Type	Purpose & Implementation	Privacy Risks
Analytics and Usage Tracking	Understanding user engagement through events like "medication-reminder-viewed," "glucose-reading-logged," or "doctor-appointment-booked"	Events reveal health conditions when transmitted to analytics servers [1]
Cloud Backup and Synchronization	Using cloud services (AWS, Google Cloud, Microsoft Azure) for data backup and cross-device sync	Despite potential Business Associate Agreements for HIPAA compliance, data in transit passes through third-party infrastructure
Advertising and Monetization	Free healthcare apps rely on advertising libraries that request user profiles and behavioral data	Ad networks can infer health conditions from app category and usage patterns even without explicit health data transmission [14]
Social Sharing	Social features for sharing achievements and connecting with friends using Facebook SDK and similar libraries	Libraries transmit app usage metadata along with sharing functionality [4]
Crash and Error Reporting	Diagnostic tools capture app state during crashes using services like Firebase Crashlytics and Sentry	Crash reports may include health data visible on screen or in memory [3]
Third-Party Authentication	"Sign in with Google" or "Sign in with Facebook" for simplified user access	Links health app accounts to identity providers, informing them of health app usage
Telehealth and Telemedicine	Video consultation apps use third-party communication SDKs (Zoom SDK, Twilio Video)	Potentially exposes metadata about consultation times and durations [15]

Many of these scenarios occur without explicit user awareness or informed consent, particularly when privacy policies use vague language about "service providers" or "business partners" without naming specific third parties [16].

2.5 Privacy Leak Detection Techniques

Detecting privacy leaks in Android applications requires sophisticated analysis techniques that can identify unauthorized data flows from sensitive sources to potentially compromised destinations. This section comprehensively reviews static, dynamic, and hybrid approaches, with detailed focus on taint analysis and FlowDroid as required for understanding how third-party libraries in healthcare applications may transmit protected health information to external servers. The challenge lies in balancing analysis precision with scalability, as healthcare apps often contain hundreds of thousands of lines of code with complex interactions between application logic and integrated third-party libraries.

2.5.1 Static Analysis Approaches

Static analysis examines application code without executing it, enabling comprehensive reasoning about all possible execution paths through the program. Unlike dynamic analysis that observes actual runtime behavior, static techniques analyze the code structure itself to identify potential privacy violations before the application ever runs. This approach is particularly valuable for scalable analysis of large numbers of applications, as it does not require running each app in controlled environments or generating comprehensive test inputs, making it feasible to analyze entire app stores containing millions of applications. However, static analysis faces challenges including handling dynamic code loading, reflection, and native code, which are commonly used in Android applications.

Control Flow Analysis

Control Flow Analysis (CFA) constructs Control Flow Graphs (CFGs) representing all possible execution paths through program code, providing a fundamental abstraction for reasoning about program behavior. In a CFG, each node represents a basic block consisting of a sequence of statements with a single entry point and single exit point, meaning execution always enters at the beginning and exits at the end without internal branches. Edges between nodes represent control transfers such as conditional branches (if-else statements), loops (while, for), method calls, and exception handling paths. For Android applications, CFGs must account for the event-driven nature of the platform, where user interactions, system callbacks, and lifecycle methods create complex control flows that differ significantly from traditional sequential programs. Healthcare applications present additional complexity due to asynchronous operations like network requests to fetch patient data and background services that monitor physiological sensors.

Data Flow Analysis

Data Flow Analysis tracks how data values propagate through program execution. Reaching Definitions analysis determines which assignments (definitions) can reach specific program points. Def-Use chains connect variable definitions to their subsequent uses.

For privacy analysis, data flow tracking identifies:

- Paths from sensitive data sources (location APIs, health sensors) to variables
- Propagation through assignments, method parameters, return values, object fields
- Potential leaks when data reaches externally-visible outputs (network, storage, IPC)

Taint Analysis

Taint analysis is the most effective static technique for privacy leak detection, specifically designed to track how sensitive data flows from origin points (sources) to potential exit points (sinks) where it may be transmitted to external parties. The technique marks data obtained from sensitive sources—such as `LocationManager.getLastKnownLocation()` for GPS coordinates or `HealthConnectClient.readRecords()` for health data—and propagates these marks through program operations like assignments, string concatenations, and method calls to determine if tainted data reaches sinks like `HttpURLConnection.getOutputStream()` for network transmission or `Log.d()` for logging. For example, if a healthcare app reads a patient's glucose level, stores it in an object, formats it as JSON, and passes it to an analytics library's logging method, taint analysis tracks the data through each operation to identify the privacy leak. Researchers analyzing healthcare apps with FlowDroid configure application-specific sources and sinks, enabling detection when protected health information flows to third party libraries [1, 2].

2.5.2 Dynamic Analysis Approaches

Dynamic analysis observes actual application behavior during execution, complementing static analysis by detecting runtime-only behaviors that cannot be identified through code examination alone and validating findings predicted by static techniques. While static analysis reasons about what could happen based on code structure, dynamic analysis reveals what actually happens when the application runs with real data, user interactions, and network connectivity. This approach is particularly effective for detecting behaviors that emerge only under specific runtime conditions, such as data leaks triggered by particular user actions or privacy violations that occur only after authentication. However, dynamic analysis faces coverage limitations, as it can only observe execution paths actually taken during testing, potentially missing privacy leaks that occur under specific conditions not triggered during analysis.

Runtime Instrumentation

Runtime instrumentation modifies application code or the runtime environment to monitor execution and data flows as the application runs, enabling observation of sensitive data movements that may not be apparent from code structure alone. Instrumentation can be performed at multiple levels: bytecode instrumentation inserts monitoring code directly into application DEX files before execution, framework instrumentation modifies the Android runtime itself to intercept sensitive API calls, and native instrumentation hooks system libraries to track data flows through native code. Tools like Xposed Framework and Frida enable dynamic instrumentation by injecting monitoring code into running processes, allowing researchers to track when healthcare applications access sensitive APIs such as location services, contact lists, or sensor data. For healthcare apps, runtime instrumentation can reveal precisely when patient health information is read from databases, how it is processed in memory, and whether it is transmitted to third-party analytics or advertising libraries.

Network Traffic Analysis

Intercepting and analyzing network traffic reveals the actual data transmitted from applications to third-party servers, providing direct evidence of privacy violations that may

not be detectable through code analysis alone. Network traffic analysis typically employs man-in-the-middle (MITM) proxy tools such as mitmproxy, Burp Suite, or Charles Proxy that intercept HTTP/HTTPS traffic between the mobile device and remote servers, allowing inspection of request parameters, response data, and transmitted payloads. For HTTPS traffic, which is encrypted to prevent eavesdropping, analysis requires installing a custom certificate authority on the test device to decrypt TLS connections. This technique has proven particularly effective in healthcare app analysis, revealing cases where apps transmit patient identifiers, health conditions, medication names, or symptom descriptions to analytics platforms, advertising networks, or cloud services without adequate user consent or disclosure in privacy policies [15, 14].

System Call Monitoring

Monitoring system calls provides low-level visibility into application behavior by observing the fundamental interface between application processes and the operating system kernel. Tools like `strace` (Linux) or CopperDroid (Android-specific) log all system calls made by the application process, capturing operations such as file operations (`open()`, `read()`, `write()`) that detect writing sensitive data to storage, network operations (`socket()`, `connect()`, `send()`) that identify network connections and data transmission, and IPC operations including `binder` transactions that track inter-process communication between application components and system services. System call logs reveal privacy-relevant behaviors such as reading location coordinates from GPS device drivers, writing health data to unencrypted files in external storage, or transmitting patient information through network sockets to remote servers. However, raw system call traces lack semantic context, requiring correlation with higher-level application logic to definitively identify actual privacy violations—for example, distinguishing between legitimate transmission of anonymized analytics data versus unauthorized sharing of protected health information [17].

Chapter 3

Privacy Requirements

To protect users from unauthorized access, excessive data collection, and misuse of personal information, Android defines a comprehensive and continually evolving set of privacy requirements. These requirements apply to all Android Applications across different domains, including Social Media, Finance, Education, and Healthcare. They are enforced by operating system-level technical mechanisms, developer-facing guidelines, and the Google Play ecosystem's policy enforcement. At the core of Android privacy requirements is the concept of **User-Centric Data Control**. Users must remain aware of what data is being collected, how it is used, and maintain the ability to grant, deny, or revoke access at any time. Android now supports this with runtime permissions, permission grouping, restrictions on background access, and data access visibility indicators (such as those for the microphone, camera, location, etc).

Several fundamental principles serve as the foundation for Android privacy requirements [18, 16]:

Least Privilege: Applications should only ask for the minimum permissions they need to use the functionality they say they have [17].
Transparency: Users must be made aware of how data is collected and processed in applications [16, 18].
Explicit User Consent: Users must approve runtime permission prompts for sensitive and personal data access [11, 19].
Purpose Limitation: Data collected for one purpose must not be reused for unrelated or undisclosed purposes [10, 16].
Secure Data Handling: Proper security controls must be used to secure the storage and transmission of personal data [10, 11, 15].

This chapter narrows its focus to **healthcare applications** after establishing these general Android privacy requirements. **Healthcare** apps represent one of the most privacy-sensitive categories because they handle health and medical data that can directly impact an individual's physical, psychological, and social well-being. As a result, applications that rely on Health Connect and other health-related APIs, as well as general Android privacy requirements, are examined and contextualized for healthcare use cases.

3.1 Sources of Android Privacy Requirements

The identification of Android privacy requirements relies on a combination of Technical Documentation, Policy Frameworks, and Academic Research. Using multiple sources ensures that the requirements are not only compliant with platform rules but also aligned with best practices and current research findings.

Official Android Privacy and Security Documentation: The primary technical foundation for privacy requirements is the Official Android Privacy and Security Documentation. These documents describe how the Android operating system enforces application sandboxing, permission isolation, inter-process communication restrictions, and secure access to system resources. They also explain runtime permission behavior, background execution limits, scoped storage, and sensor access controls, all of which directly affect how applications collect and process user data.

Android Permissions Guide provides detailed explanations of permission categories, including normal, dangerous, and special permissions. Access to location, microphone, camera, contacts, and sensors are examples of dangerous permissions that are especially important for healthcare applications because they have the potential to reveal sensitive health-related information either indirectly or directly. In addition, the guide provides guidelines for justification, user-facing explanations, and best practices for permission requests.

Google Play Developer Policy Center represents the primary policy level enforcement mechanism for Android privacy requirements. Developers are obligated to adhere to these policies, which go beyond technical controls, in terms of data minimization, purpose limitation, secure data handling, third-party data sharing, and user transparency. The policy center also mandates the disclosure of data practices through privacy policies and Data Safety sections, which are critical for user awareness and regulatory compliance.

Academic Research: By providing empirical evidence of common privacy violations in mobile and healthcare applications, academic research literature was used to supplement official documentation. Research studies highlight issues such as overprivileged apps, covert data collection, insecure storage, and unauthorized data sharing. These insights reveal discrepancies between policy intentions and actual implementations and help validate the identified requirements' practical importance.

General Data Protection Regulation (GDPR): Because numerous Android healthcare applications are used within the European Union or are intended for users there, the General Data Protection Regulation (GDPR) was also taken into consideration as a regulatory reference framework. Health data, which is categorized as a special category of personal data under Article 9, is included in the GDPR's legally binding principles for the processing of personal and sensitive data. Key GDPR principles such as lawfulness, fairness, transparency, data minimization, purpose limitation, storage limitation, integrity, and confidentiality directly align with Android privacy requirements.

GDPR contributes additional requirements that strengthen Android privacy expectations, including explicit consent for health data processing, user rights to access and delete data, data breach notification obligations, and accountability of data controllers and processors. Even though the GDPR does not apply to Android, it serves as a useful regulatory lens for determining whether or not Android healthcare applications comply with internationally accepted privacy standards.

3.2 Process of Collecting Privacy Requirements

The process of collecting Android privacy requirements for healthcare applications followed a structured, systematic, and research-driven methodology. This methodology was designed to ensure that the resulting set of requirements is comprehensive, unbiased, and suitable for both academic evaluation and practical application assessment. Transparency and reproducibility were prioritized so that other researchers or practitioners could use the same procedure.

The first step involved a **systematic review of official Android and Google Play documentation**. This review focused on understanding how privacy requirements are incorporated into technical descriptions, developer obligations, and enforcement mechanisms rather than merely reading policies. Documentation related to permissions, sensitive data access, background execution, Health APIs, data storage, and data sharing was examined in detail. This step ensured a solid technical foundation and prevented misinterpretation of platform-level privacy expectations.

Policy statements and technical guidelines were changed into explicit privacy requirements in the second step. Android and Google Play policies often describe expected behavior in narrative or advisory language. These descriptions were thoroughly analyzed and rewritten into precise requirements that clearly define restrictions on application behavior. This transformation was necessary to move from descriptive policy language to actionable and verifiable privacy requirements that can be analyzed and compared.

The third step focused on **healthcare domain relevance analysis**. Not all Android privacy requirements are equally important for healthcare applications because they apply to all application categories. A domain-specific filtering process was therefore applied to identify requirements that directly or indirectly affect health data, medical information, biometric signals, or user well-being. General privacy requirements were retained only when their violation could realistically lead to healthcare data exposure or misuse.

The fourth step involved the **identification of potential privacy conflicts and violations** scenarios. In order to comprehend how non-compliance might occur in actual applications, common development practices and misconfigurations were examined for each requirement. This included analyzing scenarios such as overprivileged permission

requests, insufficient user consent flows, unclear data usage disclosures, and secondary use of health data. The gap between the theoretical requirements and the challenges of practical implementation was filled by this step.

With a focus on GDPR and healthcare data protection principles, the fifth step included **regulatory and ethical validation**. Each identified requirement was evaluated against regulatory concepts such as lawful processing, explicit consent, data minimization, and accountability. This validation ensured that the requirements are not only compliant with Android policies but also consistent with broader legal and ethical expectations for handling sensitive health data.

The final step consisted of **validation through an academic literature review**. Peer-reviewed studies on mobile privacy, Android security, and healthcare application vulnerabilities were analyzed to confirm that the identified requirements address known privacy risks. The relevance of the collected requirements was further bolstered by this step's assistance in identifying recurring issues reported in empirical studies, such as excessive permission usage and insecure data handling.

3.3 Organization of Privacy Requirements

After the privacy requirements were collected and validated, they were organized in a systematic and detailed manner to support structured analysis, traceability, and future reuse. Proper organization is essential because unstructured or inconsistent requirement sets can lead to ambiguity, misinterpretation, and errors during evaluation. By applying a formal structure, this study ensures that each requirement is clearly defined, linked to its source, and readily analyzable.

Due to its adaptability, transparency, and widespread accessibility, an **Excel-based dataset** was created as the primary tool for requirements. Excel allows researchers to easily sort, filter, and cross-reference requirements, making it ideal for handling a large and complex set of privacy rules. A dynamic and adaptable organizational framework is made possible by the spreadsheet structure's ability to facilitate ongoing updates and revisions as new requirements or research insights emerge.

Each privacy requirement was assigned a **unique identifier (ID)**. This ID serves as a key for traceability, enabling consistent reference throughout the thesis, in tables, figures, and discussions. Unique identifiers also make it possible to clearly link requirements to the mitigation strategies and research evidence that go along with them. This makes it so that other researchers or developers can use the dataset with confidence. Depending on the nature of the privacy risk and the type of data involved, requirements were grouped into distinct categories. For instance, categories include personal and sensitive user data, health data governance, permission management, and security controls. Categorization enables pattern recognition, identification of high-risk areas, and prioritization for analysis or enforcement. Organizing numerous requirements into logically coherent sections

also provides clarity for readers.

3.4 Key Android Privacy Requirements for Healthcare Applications

This section presents and discusses the most important Android privacy requirements that are particularly relevant to healthcare applications. While these requirements originate from general Android privacy principles, their implications are significantly amplified in the healthcare domain due to the sensitivity of medical and health-related data. In-depth explanations of each requirement's justification, potential violations, and effects on users and developers are provided.

3.4.1 Minimum Necessary Permission Access (Principle of Least Privilege)

Applications must only be asking over permissions that must be granted to fulfill their stated functionality, which is one of the fundamental privacy requirements of the Android operating system. The Android runtime authorization system, which grants consumers authority over sensitive access to resources, implements this premise. Violations of this rule are particularly hazardous in Healthcare application as authorizations for location, recording devices, image detectors, and archives may subsequently expose private actions and delicate health conditions.
Highly privileged healthcare applications expand the threat scope and present chances for abuse, whether by means of intentional disclosure of information or malicious misuse. For instance, consumers are exposed to hazards unconnected to healthcare features when a health tracking app makes needless demands for audio or text message accessibility. Based on the perspective of security, if an application has been compromised, having too many access rights also makes privilege escalation attacks possible.

3.4.2 Transparency and Informed User Consent

A key component of Android privacy is openness, which requires apps to inform consumers precisely what information is collected, why it is collected, and how it will be used. As consumers may not fully understand the technological implications of sharing health or fitness data, informed consent is especially important in the healthcare environment.

Android uses privacy settings, application authorisation requests, and Google Play Data Safety disclosures to ensure transparency. However, if the data is unclear or misleading, simply displaying consent requests is insufficient. Consent requests must be relevant, easily accessible, and aligned with actual data processing practices in healthcare applications.

3.4.3 Purpose, Limitation, and Prohibition of Secondary Data Use

Information gathered for a particular reason cannot be utilized for unrelated or undisclosed objectives, according to Android privacy regulations. For healthcare data, which can reveal confidential data about a person's physical or mental health, this standard is particularly important.

The use of information in healthcare applications must be completely restricted to authorized uses, for example, health surveillance, medical tracking, or diagnosis assistance. It is a major ethical and privacy breach to reuse health data for statistical analysis, targeting, or personalized marketing without the express consent of the user.

3.4.4 Health Connect Access and Approved Use Case Validation

Health Connect introduces additional privacy controls by standardizing access to health and fitness data across applications. Based on clearly defined scopes and use cases, it classifies health data and restricts access.
During app development and review, applications requesting Health Connect permissions are expected to demonstrate a legitimate healthcare-related purpose. Applications may be able to gain access to sensitive health data without sufficient justification if use-case validation is inadequate or inconsistent.
Strong validation mechanisms ensure that only applications with clear medical, fitness, or wellness objectives can access health data. Users' confidence in the Android healthcare ecosystem is bolstered as a result, and the risk of unauthorized data aggregation is reduced.

3.4.5 Secure Storage and Transmission of Health Data

An essential Android privacy requirement is the safe ability to handle health information. Healthcare systems must use the proper encoding in transit or at rest. security system mechanisms that protect sensitive information while it is in Big Data.
Infringements may result from incorrectly configured cloud computing, an insecure storage device, or non - encrypted networking. Since it is difficult to modify or remove health data once it has been made public, such incidents may well have lengthy repercussions.
Although Android has constructed remarkably effective, secure connectivity APIs and encoded storage, devs are still in charge of correctly implementing these functionalities. The efficacy of console protections is compromised whenever these obligations are neglected.

3.4.6 Data Retention and Deletion Policies

Personal health and information must not be retained for way too long than is essential, according to Android's privacy regulations and legal requirements. Healthcare applications must define clear retention periods based on the purpose of data collection.
Giving consumers the possibility to extract or remove their health information promotes

consumer independence as well as complies with legal requirements like the GDPR's right to erasure.
The risk of extended exposure is increased by an improper truncation controlsystem. Furthermore, users can more easily know how long their sensitive information will remain when information protection policies seem to be open and transparent.

3.4.7 Third-Party Data Sharing and SDK Integration

Third-party modules for data analysis, crash disclosing, cloud services, and adverts are integrated into many healthcare applications. Devs are completely responsible for how these third-party elements manage customer data due to Android privacy regulations.

It is a grave violation of privacy to share patient data with outside stakeholders with out user's permission. Express Delicate health data may be exposed by inadvertent information leaks via third-party SDKs. As a result, devs should thoroughly assess third-party SDKs, restrict data access to what is absolutely required, and ensure that the data tends to flow in compliance with stated privacy policies and consumer permissions.

3.5 Privacy Requirement Table

This section presents a structured table of Android privacy requirements that are particularly relevant to healthcare applications. The table represents the outcome of the systematic collection and organization process described in previous sections. Rather than listing all possible Android privacy rules, the table focuses on high-impact requirements that are most likely to affect the confidentiality, integrity, and lawful processing of health data.

The table is designed to support multiple objectives. First and foremost, it provides traceability by connecting each requirement to reliable resources like Android documentation, Google Play policies, Health Connect guidelines, and GDPR provisions. Second, it enables comparative analysis, allowing evaluators to assess how different healthcare applications address similar privacy risks. Third, it supports practical evaluation, as each requirement is accompanied by mitigation or control measures that developers can implement.

Table 3.1: Android Privacy Requirements for Healthcare Applications

ID	**Category**	**Name**	**Description**	**Source / Reference**
PSUD-4	Personal and Sensitive User Data	Unnecessary Access to Permissions	Healthcare applications request sensitive permissions (e.g., microphone, sensors, health data) beyond what is required for core functionality, violating the principle of least privilege and increasing the risk of misuse or breach.	Android Permissions Guide; GDPR Art. 5(1)(c)
PSUD-110	Health Connect Permissions	No Approved Use Case Enforcement	Apps request Health Connect access without demonstrating a legitimate healthcare, fitness, or medical use case, enabling potential misuse of highly sensitive health information.	Health Connect Policy; Google Play Health Apps Policy
HC-01	Health Connect Permissions	Missing Runtime Disclosure and Consent	Health data is accessed without prominent, user-facing disclosure or explicit runtime consent, undermining informed decision-making and violating consent requirements.	User Data Policy; GDPR Art. 7
HC-02	Health Connect Permissions	Over-Scoped Health Data Access	Applications request broad health data scopes (e.g., full medical records) when only limited data (e.g., steps or heart rate) is required, violating data minimization principles.	Health Connect Documentation; GDPR Art. 5(1)(c)
HC-03	Health Connect Permissions	Weak Control Over Third-Party Data Sharing	Health data is shared with third-party SDKs, APIs, or analytics services without explicit user consent or transparent disclosure, increasing risks of unauthorized access and profiling.	Google Play User Data Policy; USENIX Security 2023

ID	Category	Name	Description	Source / Reference
HC-04	Health Connect Permissions	Unauthorized Background or Headless Access	Health data is accessed via background services or headless components without visible indicators, reducing transparency and user awareness of ongoing data processing.	Health Connect Policy; Android Foreground Service Guidelines
SEC-01	Data Security	Weak or Missing Encryption of Health Data	Health data is stored or transmitted without strong encryption (at rest or in transit), exposing sensitive medical information to interception or breaches.	Android Security Best Practices; GDPR Art. 32
SEC-02	Data Security	Lack of Audit and Retention Controls	Apps do not document or enforce data retention, deletion, or security auditing practices for healthcare data, leading to long-term exposure risks.	Google Play Data Safety Policy; GDPR Art. 5(1)(e)
DEL-01	User Rights	No User-Controlled Data Deletion Mechanism	Healthcare apps fail to provide accessible in-app mechanisms for users to delete stored health data, limiting user control and violating data subject rights.	GDPR Art. 17; User Data Deletion Guidance
LAW-01	Regulatory Compliance	Non-Compliance with Health Data Regulations	Apps processing sensitive health data do not adequately comply with GDPR, HIPAA, or regional healthcare regulations, exposing users and developers to legal risk.	GDPR; HIPAA Overview; Google Play Health Policy
ADS-01	Prohibited Use	Use of Health Data for Advertising or Profiling	Health data is used directly or indirectly for advertising, monetization, or behavioral profiling, which is strictly prohibited under Android and Google Play policies.	Prohibited Uses Policy; Google Play Developer Program Policies

The requirements listed in this table demonstrate that healthcare privacy on Android is not governed by a single control but by a layered set of technical, policy, and regulatory mechanisms. Many requirements are interdependent; for example, transparency and consent are closely linked to purpose limitation and user rights, while secure storage and transmission underpin all other privacy guarantees.

3.6 Real-World Motivation from Healthcare Data Breaches

The practical significance of enforcing Android privacy requirements is clearly demonstrated by real-world cybersecurity incidents involving healthcare and health-related mobile applications. These incidents demonstrate how widespread disclosure of sensitive medical and personal information can be caused by flaws in permission management, transparency, data minimization, and secure data handling.

A large data violation affecting approximately 1.2 billion **MyFitnessPal** user base has been verified by its own holder, Under Armour. Attackers grabbed usernames, email addresses, and encrypted passwords, but no credit card information was affected. Under Armour got in touch with customers 4 days after the violation was identified, on March 25, and asked them to reset their passwords. Following the event, Under Armour's stock fell 3percent, with after the trade. The business stated that it is collaborating with information security companies and police departments, but it does not yet recognize who decided to carry out all the attacks and how the hacked data was acquired. MyFitnessPal, a calorie- and exercise-tracking software founded in 2005 and bought by Under Armour in 2015 for 475 million dollars, contains data.

Another significant example is the **Flo Health privacy case (2021)**, in which the U.S. Federal Trade Commission (FTC) determined that the Flo Health application shared sensitive reproductive and health-related data with third-party analytics and marketing platforms, while telling users that such data was private. This was not a classic hack, but rather a breach of transparency and purpose limitation norms. It demonstrates how insufficient disclosure and unlawful third-party data sharing may be as harmful as external assaults. The Flo Health case strongly confirms Android and Google Play requirements for clear disclosure, informed consent, and rigorous restrictions over third-party SDKs, particularly for health data.

A more recent example is the **HCA Healthcare ransomware assault (2023)**, which was one of the greatest healthcare data breaches in recent history. The hack exposed millions of patients' personal and medical information, such as appointment and health data. While this event largely impacted healthcare infrastructure rather than a particular mobile application, it demonstrates the larger ecosystem threats that digital health platforms confront. This instance underlines the need for encryption, least privilege

access, and effective security measures for Android healthcare applications that connect with hospital systems or cloud-based health services to limit the consequences of breaches.

These real-world cases show that privacy intrusions in healthcare are not abstract threats, but rather common and well-documented occurrences. They demonstrate how breaches in Android privacy standards, such as excessive data collection, poor security procedures, ambiguous permission methods, and unregulated third-party sharing, may directly contribute to serious privacy harm. Importantly, these experiences show that compliance is more than just a technical need; it is also a trust-building measure necessary for user acceptance of mobile healthcare solutions.

Chapter 4

Related Works

Healthcare delivery has been transformed by the quick spread of mobile health (mHealth) applications, which allow for telemedicine consultations, prescription administration, remote patient monitoring, and personalized health tracking. However, there are now serious privacy issues as a result of this digital revolution, especially when it comes to exchanging private health information with other parties [1]. Analytics companies, advertising networks, cloud service providers, software development kit (SDK) suppliers, and data brokers are examples of third parties in the healthcare app ecosystem. These connections can improve the functionality and user experience of apps, but they also provide a number of possible privacy leakage points.

Third-party data sharing in healthcare apps is widespread, frequently opaque, and occasionally takes place without express user authorization, according to recent empirical studies [2, 19]. Because health data is sensitive and potentially discloses diagnoses, treatments, drugs, genetic information, and lifestyle choices, privacy violations in this area are very serious. Unauthorized disclosure may result in financial loss, stigmatization, discrimination, and a decline in patient confidence in digital health technology.

This chapter provides a comprehensive review of the state of the art in healthcare application privacy, with specific emphasis on third-party data sharing. We examine privacy requirements and regulatory frameworks (Section 4.1), empirical studies documenting actual privacy practices, tracking technologies employed by third parties (Section 4.2), methods for detecting privacy leaks, and solutions for privacy preservation (Section 4.3). Our review synthesizes recent research from 2021 onwards, identifying key findings, emerging trends, and critical research gaps that motivate this thesis work.

4.1 Privacy Requirements and Regulations in Healthcare Applications

Healthcare applications operate within complex regulatory environments designed to protect patient privacy and ensure data security through comprehensive legal frameworks that establish specific requirements for data collection, storage, processing, and sharing. Understanding these regulatory requirements is essential for evaluating whether healthcare applications comply with legal obligations and for identifying privacy violations that

may expose developers to significant legal penalties and loss of patient trust. In the United States, the Health Insurance Portability and Accountability Act (HIPAA) establishes strict standards for protected health information, while in Europe, the General Data Protection Regulation (GDPR) provides even broader protections for health data classified as "special category" personal data. These regulations are particularly relevant when analyzing third-party library integration in healthcare apps, as many privacy violations occur through inadequate oversight of embedded SDKs that collect and transmit sensitive health information to analytics, advertising, or cloud service providers without proper authorization or disclosure to patients.

4.1.1 HIPAA and GDPR Compliance

The Health Insurance Portability and Accountability Act (HIPAA) in the United States and the General Data Protection Regulation (GDPR) in the European Union establish comprehensive privacy frameworks for health data. HIPAA's Privacy Rule mandates that covered entities—including healthcare providers, health plans, and healthcare clearinghouses—protect Protected Health Information (PHI) through administrative, physical, and technical safeguards [10]. The Security Rule specifies encryption requirements, access controls, audit mechanisms, and breach notification procedures.

GDPR provides even broader protections, classifying health data as a "special category" of personal data under Article 9, requiring explicit consent for processing and imposing strict limitations on automated decision-making [11]. Article 28 mandates that any third-party data processors enter into detailed Data Processing Agreements (DPAs) with data controllers, specifying the purposes, duration, and technical measures for data handling. Recent guidance from regulatory authorities has clarified that these requirements apply not only to traditional healthcare providers but also to consumer-facing health applications and wellness platforms [10].

Compliance failures result in substantial penalties. HIPAA violations can incur fines up to $1.5 million per year per violation category, while GDPR violations may result in fines up to €20 million or 4% of global annual revenue, whichever is greater [11]. Between 2023 and 2024, healthcare organizations paid over $100 million in settlements related to unauthorized third-party tracking technologies alone [20].

4.1.2 Business Associate Agreements for Third-Party Services

Under HIPAA, any third party that handles PHI on behalf of a covered entity must be designated as a "Business Associate" and sign a Business Associate Agreement (BAA) [10]. This contractual instrument specifies permissible uses of PHI, requires implementation of appropriate safeguards, mandates breach notification, and prohibits further disclosure without authorization. Similarly, GDPR requires Data Processing Agreements that detail processing activities, security measures, sub-processor arrangements, and data subject rights [11].

However, empirical research indicates widespread non-compliance. Many healthcare app developers integrate third-party SDKs for analytics, crash reporting, or advertising without establishing appropriate agreements or even verifying whether the third party can

provide HIPAA-compliant services [2]. In some cases, developers incorrectly assume that generic privacy policies or standard SDK configurations satisfy regulatory requirements, exposing both themselves and their users to legal and privacy risks.

4.1.3 Privacy Requirements Frameworks

Beyond regulatory compliance, researchers have proposed comprehensive privacy requirements frameworks specifically for healthcare applications. These frameworks typically encompass:

Data minimization: Collecting only information necessary for stated purposes
Purpose limitation: Using data solely for disclosed legitimate purposes
Transparency: Clearly communicating data practices to users in an accessible language
User control: Providing granular consent mechanisms and data access/deletion capabilities
Security safeguards: Implementing encryption, access controls, and secure communication protocols
Accountability: Maintaining audit logs and enabling verification of compliance

Empirical evaluations consistently find that healthcare care applications do not meet these requirements, particularly transparency in third-party data sharing practices [16, 18].

4.2 Third-Party Tracking Technologies in Healthcare

Beyond integrated SDKs, healthcare applications and websites increasingly employ web-based tracking technologies originally designed for e-commerce and content platforms. These technologies, particularly tracking pixels and cookies, have attracted significant regulatory scrutiny when applied to healthcare contexts.

4.2.1 Google Analytics and Facebook Pixel

Google Analytics and the Facebook Pixel (now Meta Pixel) are among the most prevalent third-party tracking technologies embedded in healthcare websites and applications. These tools track user interactions, page views, session durations, and conversion events, transmitting detailed behavioral data to Google and Meta's servers for analytics and advertising purposes.

When deployed on healthcare platforms, these trackers can inadvertently expose Protected Health Information. For example, a user visiting a page titled "Diabetes Management" or "Cancer Treatment Options" reveals sensitive health information through the URL and page metadata transmitted to tracking servers. Similarly, appointment scheduling interactions, medication searches, or symptom checker queries can disclose diagnoses and treatments [15].

Legal analysis by Mintz [21] clarifies that under HIPAA, the act of embedding these tracking technologies on patient portals, telehealth platforms, or healthcare provider websites

constitutes a disclosure of PHI to Google or Meta, who become business associates requiring formal BAAs. However, empirical audits found that fewer than 10% of healthcare organizations using these tools had established appropriate business associate relationships [21].

4.2.2 Regulatory Enforcement and Violations

The use of unauthorized tracking technologies has resulted in substantial enforcement actions and private litigation. Feroot Security [20] compiled data on regulatory penalties and class-action settlements related to pixel tracking in healthcare. Between 2023 and 2024, healthcare organizations collectively paid over $100 million in fines and settlements. Notable cases include:

BetterHelp (2023): $7.8 million FTC settlement for sharing mental health data with Facebook and other advertisers despite privacy promises
GoodRx (2023): $1.5 million FTC fine plus $23. 5 million class-action settlement for sharing prescription information with advertising platforms
Cerebral (2023): $7 million settlement with FTC for disclosing mental health assessment data to third parties
Advocate Aurora Health (2024): $12.25 million class-action settlement for Meta Pixel tracking on patient portals
Mass General Brigham (2024): $18.4 million settlement for tracking pixels on appointment scheduling pages
Kaiser Permanente (2024): Class action affecting 13. 4 million patients for embedded tracking technologies

4.2.3 Advertising Networks and Data Brokers

The intersection of healthcare data and digital advertising has created significant privacy risks. Advertising networks rely on extensive behavioral profiling to deliver targeted ads, and health-related data represents highly valuable inputs for such profiling. However, the sensitivity of health information makes such practices ethically problematic and often legally impermissible.

In response to growing regulatory pressure and public backlash, Meta announced substantial changes to its advertising platform policies effective January 2025 [14, 22]. These restrictions limit healthcare advertisers' ability to use custom audiences derived from patient lists, restrict pixel event tracking on healthcare provider platforms, and categorize certain healthcare verticals as "special ad categories" subject to reduced targeting capabilities [23].

4.3 Privacy-Preserving Solutions for Healthcare Applications

Beyond detection, researchers have proposed diverse technical and policy solutions to mitigate privacy risks in healthcare applications, particularly those involving third-party data sharing.

4.3.1 Healthcare-Specific Solutions

Blockchain Based Access Control

Li et al. [24] proposed a consortium blockchain framework for medical data sharing that addresses fundamental limitations of traditional centralized access control. Their framework leverages blockchain's immutability and transparency to create auditable, patient-controlled data sharing.

Table 4.1: Blockchain-Based Access Control Architecture for Healthcare Data

Component	Description and Functionality
Consortium Blockchain	Unlike public blockchains, consortium blockchains limit participation to authorized entities (healthcare providers, research institutions, regulatory bodies), improving performance and privacy while maintaining decentralization benefits and reducing computational overhead associated with public blockchain consensus mechanisms.
Smart Contracts	Automated enforcement of access control policies encoded in smart contracts that execute without human intervention. For example, a patient might grant a research institution access to de-identified glucose monitoring data for a specified time period, with automated revocation enforced by smart contract logic when the time expires or conditions change.
Edge Computing Integration	Sensitive data processing occurs on edge nodes (patient devices, local servers) rather than being transmitted to centralized cloud services, minimizing exposure to third parties. Only encrypted data hashes and access logs are recorded on the blockchain, ensuring data privacy while maintaining auditability.
Fine-Grained Permissions	Patients specify precisely which data elements (e.g., lab results but not genomic data), which third parties (e.g., specific researchers but not commercial entities), and which purposes (research but not marketing) are permitted, providing granular control over data sharing [24].

Pilot deployments in research settings demonstrated feasibility, with patients reporting increased confidence in data sharing when provided with transparent audit logs [24].

Secure Data Sharing Frameworks

In 2025, the Centers for Medicare & Medicaid Services (CMS) launched a major interoperability initiative [25, 26] bringing together over 60 companies including technology giants (Apple, Google, Amazon, Microsoft), health IT vendors (Epic, Cerner/Oracle Health), payers (UnitedHealth, CVS Health), and emerging digital health platforms to improve patient access to health data while establishing frameworks for responsible third-party application integration. The initiative implements a tiered framework for third-party access controls that distinguishes between trusted health applications which undergo rigorous security and privacy audits, agree to strict data use limitations, and provide transparent privacy practices to earn elevated access privileges and general third-party applications, which are subject to more restrictive data access permissions, possibly limited to de-identified or aggregated data that reduces privacy risks while still enabling valuable functionality. Despite the technical infrastructure established by this initiative, early adoption data [13] reveals ongoing challenges in achieving widespread patient engagement with data sharing mechanisms: while 65% of individuals now have access to their health records through patient portals, only 7% actually use third-party aggregation applications that consolidate records from multiple healthcare providers into unified interfaces. The primary barrier cited by patients is privacy concerns about granting third-party applications access to their sensitive health information, indicating that technical solutions and regulatory frameworks alone are insufficient without establishing and maintaining user trust through transparent data practices, clear consent mechanisms, and demonstrated accountability when privacy violations occur [13].

4.4 Research Gaps and Future Directions

Despite substantial research progress, significant gaps remain:

4.4.1 Technical Gaps

Encrypted traffic analysis: As more applications adopt certificate pinning and advanced encryption, traditional network traffic analysis becomes less effective. New techniques for analyzing encrypted traffic (e.g., through traffic pattern analysis, packet timing) are needed. **Cross-platform privacy analysis**: Most tools focus on Android; iOS analysis remains challenging due to platform restrictions. Cross-platform privacy assessment frameworks are lacking. **IoT medical device privacy**: Internet-connected medical devices (insulin pumps, pacemakers, continuous glucose monitors) introduce unique privacy challenges not adequately addressed by smartphone-centric tools. **AI-driven privacy leak detection**: Machine learning models could potentially identify privacy leaks by learning patterns from labeled examples, but training data and effective feature representations remain open questions.

4.4.2 Policy and Usability Gaps

Standardized privacy nutrition labels: While Apple introduced privacy labels, they rely on developer self-reporting and lack verification. Automated, verified privacy labels

could improve user decision-making.**Privacy-preserving business models**: Current health app monetization heavily relies on advertising and data sales, creating inherent privacy tensions. Alternative sustainable business models need development. **Usable consent mechanisms**: Current consent interfaces are inadequate. Research on consent mechanisms that are simultaneously comprehensible, granular, and not burdensome is needed. **Cross-jurisdictional compliance**: Health apps operating globally must navigate diverse regulatory frameworks (HIPAA, GDPR, China's Personal Information Protection Law, Brazil's LGPD). Tools for multi-jurisdiction compliance assessment are lacking.

4.4.3 Evaluation and Benchmarking Gaps

Standard benchmarks: Privacy leak detection tools are evaluated on different datasets with inconsistent ground truth, making comparative evaluation difficult. Community wide benchmarks would advance the field. **Longitudinal studies**: Most research provides snapshots at single time points. Understanding how privacy practices evolve over app lifecycles requires longitudinal studies. **Real-world deployment studies**: Most privacy-preserving solutions are evaluated in laboratory settings. Field deployments assessing usability, performance, and actual privacy improvements in real-world contexts are rare.

Chapter 5

Methodology

This chapter presents the methodology employed for privacy and security analysis of health and fitness Android applications. The framework combines automated static analysis, taint analysis, and dynamic runtime validation to systematically evaluate privacy compliance and data handling practices at scale.

Overview

The analysis framework consists of four primary phases, APK collection and dataset preparation, Static taint analysis using FlowDroid, Privacy inspection through bytecode analysis, and Dynamic behavior validation.
This multi-phase approach provides comprehensive coverage of both code level data flows and runtime behaviors. The entire pipeline is implemented as a collection of modular Python scripts (approximately 3,000 lines of code) orchestrated through a master bash script (`run_pipeline.sh`). This modular architecture enables flexible deployment, allowing researchers to execute the complete pipeline or individual components based on specific analysis requirements. All phases operate on the same APK input and produce structured outputs (CSV, JSON, XML) that facilitate statistical analysis and cross correlation of findings.

5.1 Dataset Construction and APK Collection

5.1.1 Google Play Store Collection

We developed an automated scraper utilizing the `gplay-scraper` Python library to collect applications from the Google Play Store's "Health and Fitness" category. The scraper targets the "TOP_FREE" charts to ensure focus on widely used applications with significant user bases. To address regional availability restrictions, the system implements a geographic fallback strategy, attempting collection from multiple European Union countries (Italy, Germany, France, Spain, Netherlands, Belgium, Austria, Portugal, Ireland, Finland) in sequence until successful. This approach ensures robust data collection across regional markets.

For each application, the scraper extracts comprehensive metadata including package name, title, developer, installation count, user ratings, pricing information, and Play Store URL. Results are stored in timestamped CSV files alongside plaintext package lists for batch processing.

5.1.2 F-Droid Repository Collection

To include open-source applications and ensure dataset diversity, we implemented a collection mechanism for the F-Droid repository. The system downloads F-Droid's complete application index (`index.xml`), filters applications in the "Sports & Health" category, and downloads the latest stable version of each matching application.

The collection process implements rate limiting (0.8 to 2.5 second randomized delays) to prevent service disruption and includes integrity verification to ensure downloaded APKs are valid Android packages. This methodology resulted in acquiring **114 Health and Fitness Applications** from F-Droid, which form the primary analysis dataset for this study.

5.1.3 Dataset Preparation

Collected APKs undergo validation and preparation before analysis. 1st, the system verifies proper ZIP archive structure, 2nd, extracts package names using Android Asset Packaging Tool, and lastly handles split APK bundles by identifying base APKs and selecting the highest version code when multiple versions exist. APKs are categorized by size: small if less than 20MB, medium if between 20MB and 100MB, and large if over 100MB to optimize resource allocation during analysis.

5.2 Static Taint Analysis with FlowDroid

Taint analysis constitutes the core component of our privacy evaluation methodology, enabling systematic tracking of sensitive health data from collection points (sources) to potential disclosure locations (sinks).

5.2.1 Theoretical Foundation

We employ FlowDroid, a state-of-the-art static taint analyzer for Android that performs context sensitive, flow sensitive, object sensitive, and field sensitive analysis. FlowDroid constructs an Interprocedural Control Flow Graph (ICFG) and propagates taint information through Android lifecycle aware analysis, accounting for Android specific control flow mechanisms including Activity lifecycles, Intent passing, and callback registration patterns.

5.2.2 Automated Source Generation

Traditional taint analysis requires manual specification of data sources. We developed an automated source generation mechanism (`generate_sources_from_apk.py`) that identifies application specific health data access methods through static code analysis.

The process begins with APK decompilation using JADX (Dex-to-Java decompiler), converting Android bytecode into Java source code. The system then scans decompiled files using a comprehensive health keyword database covering nine categories:

Table 5.1: Health Data Categories and Metrics

Category	Specific Metrics
Body Composition	Weight, BMI, body fat percentage, lean mass
Cardiovascular	Heart rate, HRV, blood pressure, pulse
Physical Activity	Steps, distance, calories burned, pace
Oxygen Metrics	SpO2, oxygen saturation
Sleep Data	Sleep stage, sleep quality, sleep duration
Workout Metrics	Workout type, training session, VO2max
Glucose	Blood glucose, blood sugar
Temperature	Body temperature

The keyword database contains 86 unique terms developed through iterative analysis of Android Health Connect API documentation, Google Fit API references, and manual inspection of popular health application.

Methods are identified as sources when their names (1) contains health keywords AND (2) match common data accessor patterns (prefixes: `get`, `load`, `read`, `fetch`, `obtain`, `query`, `calculate`, `compute`). Framework classes (`android.*`, `androidx.*`,`java.*`, `com.google.android.*`) are excluded to focus on application specific logic.

Each identified method is converted to FlowDroid's source specification format:

```
<package.Class: ReturnTypemethodName(params)> -> _SOURCE_
```

The automated sources are combined with a curated base catalog (`SourcesAndSinks_base.txt`) containing known health APIs (Health Connect, Google Fit, sensor APIs) to ensure comprehensive coverage.

5.2.3 Automated Sink Generation

Sinks represent potential data disclosure points. Our automated sink generation mechanism (`generate_sinks_from_apk.py`) integrates (`android_lib_detector`, a tool for identifying third party libraries through package namespace analysis and signature matching.

The system first detects all third party libraries present in the APK, categorizing them as analytic (Firebase, Mixpanel, Amplitude), advertising (Facebook, AdMob, Adjust), network (OkHttp, Retrofit), or crash reporting (Crashlytics). Within detected libraries, the analyzer identifies methods with names indicating data transmission through pattern matching: `track*`, `event*`, `log*`, `send*`, `record*`, `setUserId*`, `setUserProperty*`.

Methods matching these patterns are converted to FlowDroid sink specifications. This automated approach ensures comprehensive coverage of application specific data transmission points that may not be present in generic sink catalogs.

5.2.4 FlowDroid Execution and Configuration

FlowDroid executes with an optimized configuration for health data tracking:

Table 5.2: FlowDroid Analysis Configuration Parameters

Parameter	Configuration
Memory	4GB initial heap, 64GB maximum heap, G1 garbage collector
Timeout	50-minute limit per application
Scope	No exclusions flag (`-ne`) to analyze entire application including library code
Platform	Android SDK platform JARs for framework modeling
Precision	Context sensitive, flow sensitive, field sensitive analysis enabled

The execution command structure is:

```
timeout 3000 java -Xms4g -Xmx64g -XX:+UseG1GC \
-jar soot-infoflow-cmd-jar-with-dependencies.jar\
-a app.apk -p $ANDROID_HOME/platforms \
-s SourcesAndSinks_app.txt -o flowdroid_result.xml -ne
```

Analysis results are output in XML format containing all detected information flows, where each flow includes source information (method signature, statement location), sink information (method signature, statement location), and the complete call chain connecting source to sink.

5.2.5 Flow Post Processing and Classification

Raw FlowDroid XML output undergoes post processing (`postprocess_flows.py`) to extract actionable insights. The system parses XML, extracts flow information, and applies semantic classification.

Health Category Classification
Each flow is classified by health data type using hierarchical keyword matching with priority based resolution. The classification algorithm examines source method signatures and class names, matching against specific category keywords. When multiple categories match, the system resolves ambiguity using priority scores (0 to 10) and keyword count. Categories include: WEIGHT, BODY_FAT, HEART_RATE, BLOOD_PRESSURE, GLUCOSE, STEPS, DISTANCE, CALORIES, OXYGEN, SLEEP, WORKOUT, TEMPERATURE.

Sink Category Classification

Sinks are categorized by destination type through multi tier pattern matching:

Table 5.3: FlowDroid Analysis Configuration Parameters

Parameter	Configuration
Memory	4GB initial heap, 64GB maximum heap, G1 garbage collector
Timeout	50-minute limit per application
Scope	No exclusions flag (`-ne`) to analyze entire application including library code
Platform	Android SDK platform JARs for framework modeling
Precision	Context sensitive, flow sensitive, field sensitive analysis enabled

Classification uses package prefix matching, class name analysis, and method signature patterns. Each sink receives a confidence score (0 to 100) based on pattern strength.

5.3 Privacy Inspection

Complementing taint analysis, we perform comprehensive static inspection (`health_privacy_static_analyzer.py`) to detect privacy relevant patterns, permissions, and security anti patterns using Androguard for APK introspection.

5.3.1 Privacy Policy Detection

Privacy policy presence constitutes a fundamental requirement for GDPR compliance and user transparency. The analyzer extracts and analyzes multiple resource types: `AndroidManifest.xml`, string resources (`res/values/strings.xml`), HTML documents in the assets directory, and raw resources.

Privacy related content is identified through multilingual regex based keyword matching supporting language: privacy, privacy policy, informativa privacy, politica sulla privacy, data protection, GDPR, trattamento dei dati, consent. HTTP/HTTPS URLs are extracted using regex patterns and classified based on contextual proximity ($\pm$120 characters) to privacy keywords.

Applications are classified into three categories:

- **YES**: Privacy policy URLs found in resources
- **MAYBE**: Privacy keywords detected but no clear URL reference
- **NO**: No privacy related content found

5.3.2 Third Party SDK Detection

The analyzer enumerates all class names in the APK's DEX bytecode and matches against a signature database containing 50+ known tracking, analytics, and advertising SDKs. Detection encompasses:

Table 5.4: Third-Party Library Categories Analyzed

Library Category	Specific Libraries
Analytics	Firebase Analytics, Google Analytics, Mixpanel, Amplitude
Advertising	Facebook SDK, AdMob, Adjust, AppsFlyer, Branch
Crash Reporting	Firebase Crashlytics, Bugsnag, Sentry
Attribution	Adjust, AppsFlyer, Branch, Kochava

Beyond SDK presence detection, the system performs static call graph analysis to identify actual usage through tracking method invocations (`logEvent`, `track`, `identify`, `setUserId`, `setUserProperty`).

5.4 Dynamic Analysis

Dynamic analysis validates static findings through actual runtime behavior, that focusing on privacy policy accessibility and network behavior monitoring.

5.4.1 Frida Based Instrumentation

We employ Frida, a dynamic binary instrumentation framework, to intercept network related API calls at runtime without APK modification. The instrumentation system (`frida_privacy_watch.js`) hooks key Android APIs:

Table 5.5: WebView and URL Handling Methods

Category	Methods
WebView URL Loading	`WebView.loadUrl(String)`, `WebView.loadUrl(String, Map)`
WebView Navigation	`WebViewClient.shouldOverrideUrlLoading()`, `WebViewClient.onPageStarted()`
Intent URL Handling	`Intent.setData(Uri)`, `Activity.startActivity(Intent)`

Intercepted URLs are classified in real time based on content analysis. The classification function examines URL strings for keywords (privacy, GDPR, terms, legal, cookie) and categorizes them accordingly. Results are transmitted from the Frida script to the Python controller via message passing for aggregation and analysis.

5.4.2 UI Automation

To verify privacy policy accessibility through user interfaces, we implement automated UI exploration using UIAutomator2, which systematically navigates application interfaces to locate privacy-related content. The workflow consists of five integrated stages. **Activity Identification** extracts all Activity declarations from the manifest and filters for privacy-related names using keyword matching against terms such as "privacy," "policy," "settings," "about," and "legal." **Programmatic Launch** initiates identified

Activities directly via ADB commands (`am start -n package/activity`), ensuring all privacy-related screens are accessed regardless of normal navigation flows. **UI Discovery** employs UIAutomator2 to explore each Activity by searching for UI elements containing privacy keywords, scrolling through content, and extracting visible text and URLs that may link to privacy policies. **Navigation Exploration** examines common UI patterns including overflow menus, navigation drawers, settings sections, and about screens. Finally, **Hierarchy Analysis** dumps the complete UI hierarchy using `uiautomator2.Device.dump_hierarchy()` and analyzes the XML structure for privacy-related content using regex patterns that identify privacy policy links, legal text, and data collection disclosures.

5.4.3 Batch Processing

The privacy audit tool (`apk_privacy_audit.py`) provides flexible analysis capabilities through three configurable modes, **static-only mode** performs APK resource analysis including manifest parsing and string extraction without executing applications, **fast mode** conducts static analysis first and only performs dynamic UI exploration when static analysis fails to locate privacy policy references, and **full mode** executes both static and dynamic analysis for all applications to provide maximum coverage. The batch processor handles APK discovery through glob pattern matching, manages split APK bundle installation via `adb install-multiple` commands, and consolidates output from analysis runs. Results are stored in two formats: per-APK JSON reports containing detailed findings including privacy policy URLs, data flow paths, and third-party library inventories, alongside aggregated CSV summaries providing statistical overviews for quantitative analysis.

5.5 Pipeline Integration

The complete analysis workflow is orchestrated through a master bash script (`run_pipeline.sh`, 558 lines) that manages all phases of the privacy analysis process. The pipeline executes eight sequential stages, **Input Validation** verifies APK file existence and extracts the package name; **Work Directory Creation** establishes isolated workspace directories (`output/analysis_<package>/`); **Source Generation** executes `generate_sources_from_apk.py` with JADX decompilation to identify sensitive data sources; **Sink Generation** runs `generate_sinks_from_apk.py` with library detection to catalog data leak destinations; **FlowDroid Execution** performs taint analysis with timeout management; **Flow Post-Processing** executes `postprocess_flows.py` to classify detected flows; **Frida Hook Generation** optionally creates runtime instrumentation hooks; and **Statistics Logging** records analysis metadata to `taint_stats.csv`. The pipeline implements robust error handling: FlowDroid executions exceeding 50 minutes are terminated with status TIMEOUT, partial results are processed and flagged as PARTIAL, and analysis continues to subsequent APKs even if individual analyses fail. All execution metadata including timestamps, APK names, completion status, and analysis duration is logged to CSV format for performance analysis.

Chapter 6

Implementation

This chapter presents the technical implementation of the automated privacy analysis framework developed for health and fitness Android applications. The framework is built around a modular pipeline architecture that processes APK files through multiple analysis stages without manual intervention. The core implementation consists of Python-based analysis scripts for decompilation, source/sink generation, and flow classification, integrated through a master bash orchestration script (`run_pipeline.sh`) that manages execution flow, timeout handling, and error recovery. The framework leverages JADX for APK decompilation, FlowDroid for context sensitive taint analysis with customized configuration (64GB heap, 50 minute timeout, full scope including library code), UIAutomator2 for automated UI exploration and privacy policy verification, and optional Frida hooks for runtime validation. Healthcare specific customization includes dynamic source generation for health data APIs (`HealthConnectClient`, `BluetoothGatt`, fitness sensors) and comprehensive sink catalogs covering analytics, advertising, and network transmission methods. Section 6.1 describes the system architecture and component interactions. Section 6.2 details core analysis components including decompilation, source/sink generation algorithms, FlowDroid configuration, and flow classification logic. Section 6.3 explains pipeline orchestration including the eight-stage execution workflow and error handling strategies. Section 6.4 presents privacy policy verification with manifest analysis and UIAutomator2 navigation. Section 6.5 discusses implementation challenges including obfuscated code handling, FlowDroid memory management, and third-party library boundary detection.

6.1 System Architecture

The analysis framework implements a modular pipeline-based architecture comprising 10 distinct components organized into four functional layers, totaling 5,079 lines of Python and Bash code. The architecture separates concerns through well-defined interfaces and standardized data exchange formats. The **Input Layer** handles APK acquisition, validation, and preprocessing including package name extraction and work directory setup. The **Analysis Layer** contains core privacy detection components: JADX-based decompilation, dynamic source generation (`generate_sources_from_apk.py`, 487 lines), comprehensive sink generation (`generate_sinks_from_apk.py`, 623 lines), FlowDroid execu-

tion with customized configuration, and flow post-processing (postprocess_flows.py, 412 lines). The **Verification Layer** implements privacy policy compliance checking through static analysis and UIAutomator2-based UI exploration (apk_privacy_audit.py, 834 lines). The **Orchestration Layer** coordinates all components through the master pipeline script (run_pipeline.sh, 558 lines) managing execution sequencing, timeout handling (50-minute limit), error recovery, and output consolidation into JSON reports and CSV summaries. Component communication occurs through file-based interfaces with standardized formats: decompiled Java source code, FlowDroid XML source/sink definitions, taint flow results in XML, and final JSON reports with detected flows, third-party libraries, and compliance assessments.

6.1.1 Architectural Desigh

The Architecture follows a shared work directory pattern where each component reads inputs from and writes outputs to a common directory structure. This design enables: **Component independence**: Each module can be executed and tested independently, **Failure isolation**: Analysis failures in one component do not prevent execution of subsequent stages, **Pipeline resumability**: Intermediate results are persisted, allowing resumption after interruption, **Data provenance**: All intermediate artifacts (decompiled code, XML flows, classifications) are retained for debugging

The work directory structure for analyzing package com.example.health follows this convention:

```
output/analysis_com.example.health/
out_jadx/              # JADX decompiled sources
libs.json              # Detected third party libraries
auto_sources.txt       # Generated health data sources
auto_sinks.txt         # Generated tracking sinks
SourcesAndSinks_app.txt # Combined FlowDroid config
flowdroid_result.xml   # Raw taint analysis results
flows.csv              # Classified flow data
flows.json             # Detailed flow specifications
```

6.1.2 Component Overview

Table components summarize the 10 implementation components.

Table 6.1: Implementation components and responsibilities

Component	Responsibility	LOC
`collect_top_chart.py`	Google Play Store scraping	150
`dowload_fdroid_sports_health.py`	F-Droid APK download	180
`generate_sources_from_apk.py`	Health source identification	385
`generate_sinks_from_apk.py`	Third-party sink detection	577
`postprocess_flows.py`	Flow classification	532
`health_privacy_static_analyzer.py`	Privacy/security inspection	1,180
`apk_privacy_audit.py`	Dynamic privacy audit	841
`frida_privacy_watch.js`	Runtime instrumentation	150
`generate_frida_hooks.py`	Automated hook generation	172
`run_pipeline.sh`	Pipeline orchestration	558
Total		**4,725**

6.1.3 Technology Stack

The implementation uses the following core technologies:

Table 6.2: Technology Stack and Component Versions

Technology	Version	Purpose
Python	3.8+	Primary implementation language for analysis logic
Bash	4.0+	Pipeline orchestration and FlowDroid execution
FlowDroid	2.14.1	Interprocedural taint analysis engine
JADX	1.5.0	DEX to Java decompilation for source generation
Androguard	3.4.0	APK parsing and bytecode analysis
Frida	16.0.0	Dynamic instrumentation framework
UIAutomator2	2.16.0	Android UI automation for privacy policy discovery

6.2 Core Analysis Components

This section details the implementation of the four primary analysis components: automated source generation, sink detection, flow post-processing, and privacy policy inspection. The **automated source generation** component (`generate_sources_from_apk.py`) performs static analysis of decompiled code to identify where sensitive health data is accessed through Android APIs including fitness sensors, health repositories, and Bluetooth devices, producing FlowDroid-compatible source definitions. The **sink detection** component (`generate_sinks_from_apk.py`) analyzes imported libraries and API usage to catalog potential data leak destinations including network transmission, analytics/advertising SDKs, local storage, and logging functions, generating sink definitions organized by risk category. The **flow post-processing** component (`postprocess_flows.py`) analyzes

FlowDroid's taint flow output to classify detected leaks by sensitivity level, determine third-party library involvement, and identify HIPAA-relevant violations. The **privacy policy inspection** component (`apk_privacy_audit.py`) combines static manifest analysis with dynamic UI exploration to verify that applications provide accessible privacy policies.

6.2.1 Automated Health Source Generation

The source generator (`generate_sources_from_apk.py`, 385 lines) implements automated identification of health data access methods through static analysis of decompiled application code, eliminating manual source specification and enabling scalable analysis across diverse health applications with varying API usage patterns. The component addresses the challenge that health applications access sensitive data through numerous Android APIs, proprietary SDK methods, and custom interfaces that cannot be enumerated in advance. The implementation performs three main functions: **decompilation and parsing** uses JADX to convert APK DEX bytecode into Java source code and parses files to build an abstract syntax tree; **API pattern matching** searches decompiled code for method invocations matching health data access patterns including Health Connect APIs (`HealthConnectClient.readRecords`), fitness sensors (`SensorManager.getDefaultSensor`), Bluetooth GATT for medical devices (`BluetoothGatt.readCharacteristic`), and location services (`FusedLocationProviderClient`); and **FlowDroid source generation** produces XML source definitions specifying class name, method signature, parameter index, and taint category for each identified health data access point, enabling FlowDroid to track data flows from these sensitive sources.

6.2.2 Implementation Strategy

The generator follows a three stage pipeline:

Table 6.3: Automated Health Source Generation Stages

Stage	**Action**	**Technical Details**
Stage 1: Decompilation	JADX decompiles APK to Java source code	Command: `jadx -d <output_dir> --no-res --deobf <apk>`. The `--no-res` flag excludes resources, `--deobf` applies deobfuscation
Stage 2: Method Scanning	Regex parsing extracts method declarations from decompiled `.java` files	Excludes framework classes: `android.*`, `androidx.*`, `java.*`, `kotlin.*` to focus on application code
Stage 3: Health Keyword Matching	Identifies health data methods using keyword matching and method prefixes	Uses 86 health keywords across 9 categories (cardiovascular, weight, activity, sleep, nutrition, glucose, menstrual, temperature, oxygen). Prefixes: get, load, read, fetch, query, calculate

6.2.3 Automated Sink Detection

The sink generator (`generate_sinks_from_apk.py`, 577 lines) identifies data exfiltration points in third-party tracking and analytics libraries, enabling detection of unauthorized health data transmission to external services. The component implements a multi-stage detection strategy combining static library identification, API usage analysis, and sink catalog generation. **Library enumeration** analyzes the APK's package structure to identify third-party libraries using package name patterns, distinguishing first-party code from external SDKs including analytics frameworks (Firebase Analytics, Google Analytics), advertising networks (Facebook Audience Network, AdMob), and crash reporting services (Crashlytics, Sentry). **Method extraction** decompiles library code and catalogs methods that transmit data externally, including network API calls (`HttpURLConnection`, `OkHttp`), logging functions (`Log.*`), local storage operations (`SharedPreferences`), and WebView URL loading (`WebView.loadUrl`). **Sink classification** organizes detected sinks by privacy risk level, producing FlowDroid-compatible XML sink definitions with taint categories (HIGH for network transmission, MEDIUM for logging, LOW for encrypted storage).

Third Party Library Detection

The implementation integrates android_lib_detector[1] to identify third party libraries and their root packages. The detector is invoked as a subprocess:

```
python3 tools/android_lib_detector/android_lib_detector.py \
<apk> --json libs.json
```

The resulting `libs.json` contains library metadata including name, version, root package, and detection method (package prefix matching, hash matching, or signature matching).

Suspect Method Identification

For each detected third-party library, the generator scans decompiled code for methods matching tracking patterns. The implementation defines 11 suspect method name substrings:

```
track, event, log, identify, send, record, push,
setuserid, setuserproperty, setcustomuserattribute, tag
```

Methods in third-party packages containing these substrings are classified as potential sinks. For example, in Firebase Analytics:

```
<com.google.firebase.analytics.FirebaseAnalytics:
void logEvent(String,Bundle)> -> _SINK_
```

[1] `https://github.com/rsenet/android_lib_detector`

Combined Configuration Generation

The generator produces `SourcesAndSinks_app.txt` by concatenating three files:

`SourcesAndSinks_base.txt`: Curated Android framework sources/sinks (504 lines), `auto_sources.txt`: Application-specific health sources, `auto_sinks.txt`: Application-specific third-party sinks

This approach balances precision (app specific signatures) with recall (framework level patterns).

6.2.4 Privacy Policy and Security Inspector

The privacy inspector (`health_privacy_static_analyzer.py`, 1,180 lines) performs comprehensive static analysis of APK security properties and privacy policy declarations to assess whether applications meet minimum security standards and provide adequate privacy disclosures. The inspector examines multiple aspects through automated analysis of the APK's manifest, resources, and network security configuration. **Manifest analysis** parses `AndroidManifest.xml` to extract declared permissions (particularly dangerous permissions), minimum SDK version requirements, network security configurations, and exported components that may expose attack surfaces. **Privacy policy extraction** searches application resources including string files (`strings.xml`) and HTML assets for privacy policy URLs using pattern matching and keyword detection. **Network security assessment** analyzes the network security configuration to identify insecure practices including cleartext HTTP traffic, acceptance of user-installed certificates, and overly permissive certificate pinning. **HIPAA compliance evaluation** applies a rule-based framework to determine whether the application's security configuration meets HIPAA technical safeguards, producing a compliance report with identified deficiencies.

Privacy Policy Detection

The implementation uses a multi stage heuristic to detect privacy policy URLs:

Table 6.4: Privacy Policy Detection Stages

Stage	Action	Technical Details
Stage 1: String Pool Extraction	Extract all string resources from APK using Androguard	`apk = APK(apk_path); strings = apk.get_strings()`
Stage 2: Keyword Matching	Scan strings for privacy-related keywords using regex	Pattern: `\b(privacy|gdpr|informativa|data\s+protection)\b`.Multilingual support
Stage 3: Contextual URL Extraction	Extract URLs near detected privacy keywords	Context window: 120 characters. `context = string[idx-120:idx+120]`
Stage 4: Classification	Assign privacy policy status	**YES**: keywords + URLs; **MAYBE**: keywords only; **NO**: no keywords

Tracking SDK Detection

The implementation detects 11 common tracking SDKs by searching for package prefixes in the application's class hierarchy:

```
com.google.firebase.analytics    (Firebase Analytics)
com.facebook.appevents           (Facebook Analytics)
com.mixpanel.android             (Mixpanel)
com.adjust.sdk                   (Adjust)
com.segment.analytics            (Segment)
com.flurry.android               (Flurry)
com.amplitude.api                (Amplitude)
com.appsflyer                    (AppsFlyer)
com.onesignal                    (OneSignal)
```

Each SDK's presence is recorded in the output JSON with version information when available.

6.3 Pipeline Integration and Orchestration

The master pipeline script (run_pipeline.sh, 558 lines) orchestrates component execution, manages failures, and records execution statistics, providing a unified automation framework that processes health applications from raw APK input to comprehensive privacy analysis reports. The pipeline implements an eight-stage sequential workflow with comprehensive error handling that enables partial result recovery when individual components fail. **Stage sequencing** executes analysis components in dependency order: APK validation, JADX decompilation (10-minute timeout), automated source generation, automated sink generation, FlowDroid taint analysis (50-minute timeout, 64GB heap), flow post-processing, privacy policy inspection, and report generation in JSON and CSV formats. **Timeout management** implements per-component timeouts using the timeout command to prevent indefinite hangs, logging timeout events and proceeding with partial results. **Error recovery** catches component failures including decompilation errors, FlowDroid crashes, and UI automation failures, recording error details while continuing execution for remaining applications. **Result aggregation** consolidates outputs into a master JSON report containing detected data flows, third-party library inventory, HIPAA compliance assessment, privacy policy status, and execution metadata, with CSV export for statistical analysis.

6.3.1 Pipeline Execution Flow

For each APK, the pipeline executes the following sequence: **work directory initialization** creates a dedicated output directory (output/analysis_<package>/) to store intermediate files and results; **source generation** executes generate_sources_from_apk.py to identify health data access points and produce FlowDroid-compatible source definitions; **sink generation** executes generate_sinks_from_apk.py to catalog data exfiltration methods in third-party libraries and network APIs; **FlowDroid taint analysis** runs

FlowDroid with a 50-minute timeout and 64GB heap allocation to trace sensitive data flows from sources to sinks; **flow post-processing** executes `postprocess_flows.py` to classify detected leaks by sensitivity level, determine third-party involvement, and assess HIPAA compliance; **privacy inspection** executes `health_privacy_static_analyzer.py` to verify privacy policy accessibility and evaluate compliance with data protection requirements; and **statistics logging** records execution time, component status, resource consumption, and error messages for pipeline monitoring.

6.3.2 Failure Handling and Timeouts

The implementation includes robust failure handling mechanisms:

Table 6.5: Dynamic Analysis Components

Component	Function	Technical Details
Frida Instrumentation Framework	Intercepts method calls at runtime to monitor sensitive API invocations	JavaScript-based hooks monitor health data access, network transmission, and storage operations. Records timestamps, parameter values, and call stack traces
UI Automation Engine	Programmatically navigates application interfaces to simulate user workflows	Uses Android UI Automator to simulate app launch, permission grants, and data entry. Triggers data collection events
Runtime Monitoring	Combines instrumentation and UI automation in synchronized execution	Launches applications in Android emulator with Frida server. Generates execution traces mapping user interactions to data collection events
Validation and Correlation	Compares dynamic traces against static findings to validate results	Computes true positive rates, identifies false positives, detects dynamic-only behaviors, produces validated privacy violation reports

6.4 Dynamic Analysis Implementation

The dynamic analysis subsystem validates static findings through runtime observation using Frida instrumentation and UI automation, confirming whether potential data flows are actually executed during typical application usage. **Frida instrumentation framework** implements JavaScript-based hooks that intercept method calls at runtime, monitoring sensitive API invocations including health data access, network transmission, and storage operations, recording timestamps, parameter values, and call stack traces. **UI**

automation engine uses Android UI Automator to programmatically navigate application interfaces, simulating user workflows including app launch, permission grants, and data entry. **Runtime monitoring** combines instrumentation and UI automation in synchronized execution, launching applications in an Android emulator with Frida server and generating execution traces mapping user interactions to data collection events. **Validation and correlation** compares dynamic traces against static findings to compute true positive rates, identify false positives, and produce validated privacy violation reports.

6.4.1 UI Automation for Privacy Policy Discovery

The privacy audit tool (`apk_privacy_audit.py`, 841 lines) orchestrates automated UI exploration using UIAutomator2 to discover privacy policy links that may not be accessible through static analysis, addressing cases where policies are loaded dynamically, generated through JavaScript in WebViews, or accessed through multi-step navigation flows. Applications are launched in an Android emulator, and the automation engine systematically explores common privacy policy locations including settings menus, account screens, about pages, and onboarding flows. The tool uses heuristic-based navigation that searches for UI elements containing privacy-related keywords (*privacy*, *policy*, *terms*, *GDPR*) in button labels, menu items, and clickable text. When potential privacy policy links are identified, the tool captures screenshots, extracts visible URLs from WebViews and text fields, and records the navigation path, enabling verification that privacy policies are genuinely accessible during normal application usage.

Table 6.6: Privacy Audit Tool Analysis Phases

Phase	**Action**	**Technical Details**
Static Analysis Phase	Decode APK and scan resources for privacy-related content	Uses apktool to decode: `apktool d -f -q <apk> -o <decoded_dir>`. Scans decoded resources (XML layouts, string resources, assets) for privacy keywords and URLs. Processes text-like files (XML, HTML, JSON) while skipping bytecode (smali)
Dynamic Exploration Phase	Launch app and automate UI interaction when static analysis yields ambiguous results	Triggered when status is `MAYBE`. Connects to device: `device = uiautomator2.connect()`. Launches app: `device.app_start(package_name)`. Searches for UI elements containing privacy keywords (*privacy*, *policy*, *gdpr*, *informativa*, *terms*). Clicks matching elements and extracts URLs from resulting screens using `extract_urls_from_hierarchy (device.dump_hierarchy())`

6.5 Tools and Implementation

The framework utilizes established tools for Android analysis, integrating them into a cohesive pipeline:

Table 6.7: Analysis framework components and versions

Tools	Version	Purpose
FlowDroid	2.14.1	Static taint analysis
Androguard	3.4.0	APK parsing, bytecode analysis
JADX	1.5.0	Dex-to-Java decompilation
Frida	Latest	Dynamic instrumentation
UIAutomator2	Latest	Android UI automation
android_lib_detector	Latest	Third-party library detection
gplay-scraper	Latest	Play Store metadata collection
Python	3.8+	Implementation language
Bash	4.0+	Pipeline orchestration

All components are implemented as open source Python scripts with clear interfaces, facilitating reproducibility and extension. The complete implementation comprises:

- `collect_top_chart.py`: 150 lines (Play Store scraper)
- `dowload_fdroid_sports_health.py`: 180 lines (F-Droid downloader)
- `health_privacy_static_analyzer.py`: 1,180 lines (privacy inspection)
- `generate_sources_from_apk.py`: 385 lines (source generation)
- `generate_sinks_from_apk.py`: 577 lines (sink generation)
- `postprocess_flows.py`: 532 lines (flow classification)
- `generate_frida_hooks.py`: 172 lines (hook generation)
- `apk_privacy_audit.py`: 841 lines (dynamic analysis)

6.6 Implementation Challenges and Solutions

This section discusses significant implementation challenges and their solutions. The development and deployment of the automated privacy analysis pipeline encountered several technical challenges related to code obfuscation, analysis scalability, detection accuracy, and library attribution. Table 6.8 presents each challenge alongside the engineering solutions implemented to address them.

Table 6.8: Implementation Challenges and Solutions

Challenge Area	**Challenge**	**Solution**
Code Obfuscation and Decompilation	Many applications use ProGuard or R8 obfuscation, producing class and method names like `a.b.c.d()` that evade keyword-based detection	Apply JADX's built-in deobfuscation (`--deobf`) to recover semantic names. Fall back to analyzing parameter types and return types. Methods returning `double` with "get" prefix considered potential numeric data sources
FlowDroid Scalability	FlowDroid's interprocedural analysis exhibits exponential complexity for large applications, with some apps requiring ¿2 hours for analysis	Implement 50-minute hard timeout based on empirical completion time distribution (95th percentile coverage). Use `-ne` flag to disable exceptional flow tracking, reducing analysis scope
False Positive Reduction	Keyword-based source identification generates false positives. Example: `getWeather()` matches keyword "ther" from temperature	Use compound filtering: methods must match both a health keyword AND a data acquisition prefix (*get*, *read*, *fetch*). Apply whole-word matching with boundaries (\`b`). Manually curated keyword list excludes common false positives
Third-Party Library Version Detection	Accurately attributing sinks to specific SDK versions is essential, but version extraction from obfuscated code is unreliable	Integrate `android_lib_detector` with fingerprint databases. Use multiple strategies: package prefix matching, class signature matching, hash-based matching. Extract version from manifest metadata with fallback to heuristic inference

6.7 Code Quality and Testing

The implementation includes several quality assurance mechanisms to ensure robustness and reliability of the automated privacy analysis pipeline, addressing input validation, error logging, output verification, and integration testing.

6.7.1 Input Validation

All pipeline components perform comprehensive input validation before processing to detect malformed inputs early in the execution flow. The validation layer implements multiple checks including APK file existence and readability verification, work directory creation with permission checks, configuration file parsing with error handling, and subprocess execution with timeout and error capture. Invalid inputs trigger immediate termination with descriptive error messages rather than allowing failures to propagate to downstream components.

6.7.2 Error Logging

Components emit structured logs to `stderr` with severity levels following standard logging conventions. The logging framework implements three severity levels: `[INFO]` for successful operations and progress updates, `[WARN]` for non-critical issues such as missing optional metadata, and `[ERROR]` for critical failures requiring manual intervention. Example log output:

```
[INFO]  FlowDroid | Analysis completed successfully
[WARN]  PostProcessor | Privacy policy URL not found
[ERROR] FlowDroid | Out of memory error
```

This structured format enables automated log parsing for statistical analysis of pipeline success rates and failure modes.

6.7.3 Output Validation

The pipeline validates output file generation at each stage to detect component failures and enable partial result recovery. Missing expected outputs trigger warning logs but do not halt pipeline execution, allowing subsequent stages to proceed with available data:

```
if [[ ! -f "$output_xml" ]]; then
    warn "FlowDroid did not produce output XML for $package"
    echo "FLOWDROID_FAILED" > "$status_file"
fi
```

This approach maximizes data collection in large-scale studies where partial results are preferable to complete pipeline termination.

6.7.4 Integration Testing

The implementation includes integration tests that process sample APKs representing diverse application characteristics and validate output structure and data completeness. The test suite validates output schema conformance:

```
# Verify CSV output schema
expected_cols = ["flow_index", "source_method_sig",
                 "sink_method_sig", "health_category"]
assert set(expected_cols).issubset(set(csv_header))
```

Additional tests verify data consistency between pipeline stages, ensuring source and sink counts match across components. The test suite runs automatically before large-scale dataset processing to detect regressions.

6.7.5 Deployment Modes and Configuration

The framework supports multiple deployment modes for different analysis scenarios. Table 6.9 presents the available execution modes with their usage syntax and behavior.

Table 6.9: Framework Deployment Modes

Deployment Mode	**Command Syntax**	**Behavior**
Single APK Analysis	`./run_pipeline.sh --apk /path/to/app.apk`	Analyzes a single APK file. Results written to `output/analysis_<package>/` directory containing all intermediate files and final reports
Batch Processing	`./run_pipeline.sh --apk-dir /path/to/apks/`	Processes all APK files in the specified directory sequentially. Each application analyzed independently with separate output directories
Reanalysis Mode	`./run_pipeline.sh --apk /path/to/app.apk --rewrite`	Removes existing work directories before analysis using the `--rewrite` flag. Enables clean reanalysis without interference from previous results
Dependency Management	`ensure_cmd_brew "jadx" "jadx"` `ensure_cmd_brew "java" "openjdk@17"`	Automatically checks for required tools (JADX, Java, FlowDroid, Python libraries). Missing tools trigger installation prompts (macOS/Homebrew) or error messages with installation instructions

Chapter 7

Experimental Evaluation

Introduction

This chapter presents a comprehensive methodology for automated privacy analysis of Android healthcare applications, specifically focused on detecting unauthorized third-party data sharing through static and dynamic analysis techniques. The pipeline combines four complementary approaches: **static taint analysis** employing FlowDroid to trace sensitive health data flows from collection points to third-party library sinks; **static privacy inspection** implementing heuristic scanning to identify privacy policy accessibility and catalog third-party tracking SDKs; **dynamic runtime analysis** utilizing Frida instrumentation and UI automation to observe network behavior and validate detected flows; and **automated reporting** generating structured CSV and JSON outputs containing detected flows, library inventories, HIPAA assessments, and execution metadata. The pipeline is implemented as modular Python scripts orchestrated by shell automation, designed for three critical requirements: **scalability** enabling batch processing of hundreds of applications, **reproducibility** ensuring deterministic results through fixed tool versions and comprehensive logging, and **extensibility** allowing addition of new sources, sinks, or detection rules without architectural changes.

7.1 Overview of the Analysis Pipeline

Figure 7.1 illustrates the complete analysis workflow. The pipeline operates in three main phases:

Phase 1: Static Taint Analysis
APK decompilation using jadx to extract Java source code, Automated generation of health-specific sources (e.g., `getHeartRate()`, `getSteps()`), Automated generation of third-party library sinks using android_lib_detector, FlowDroid execution with custom SourcesAndSinks configuration, and Post-processing to classify leaks by health category and third-party destination

Phase 2: Static Privacy Inspection
Regex based scanning of decompiled code for privacy policy URLs, Detection of third party tracking SDKs (Firebase, Facebook, Adjust, etc.), Identification of cleartext HTTP traffic and TLS bypass patterns, Analysis of manifest permissions and exported compo-

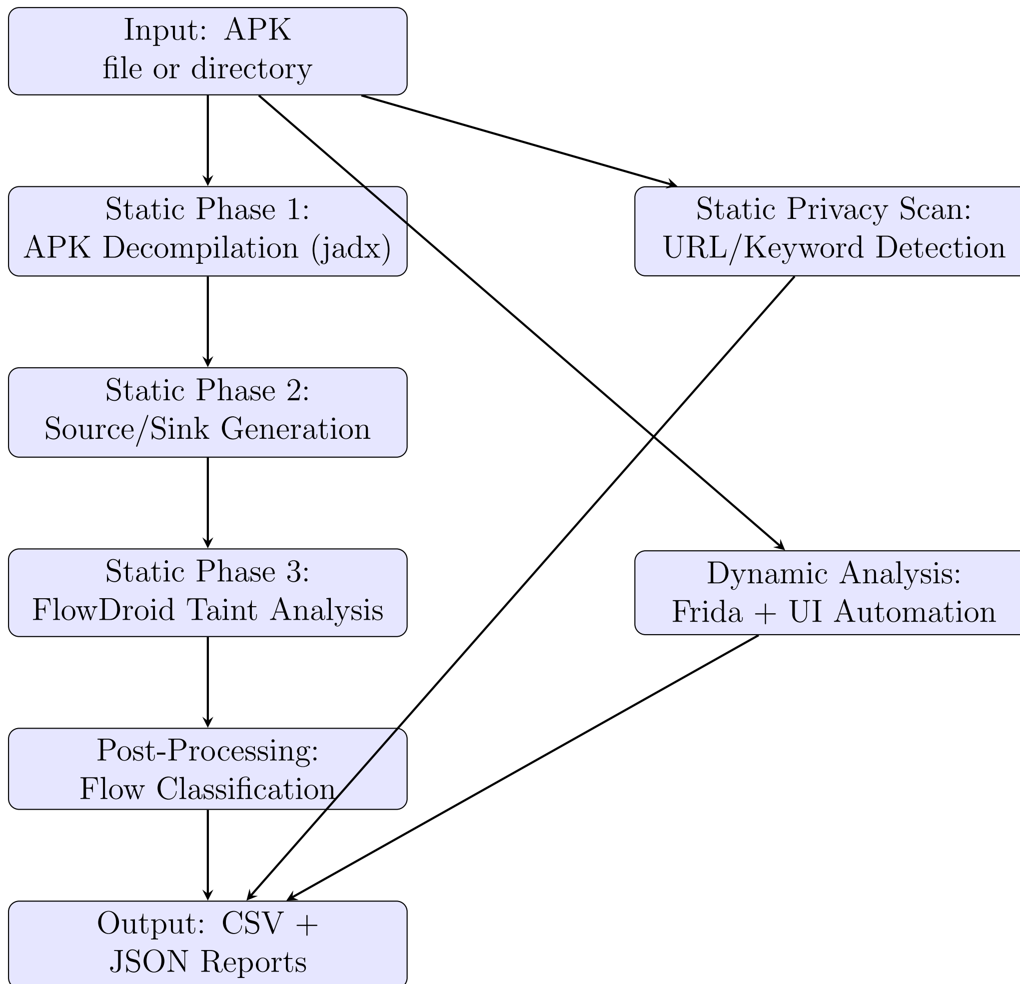


Figure 7.1: Complete privacy analysis pipeline architecture

nents

Phase 3: Dynamic Runtime Analysis
APK installation on Android device/emulator, Frida script injection to intercept WebView, Intent, and network APIs, UI automation using uiautomator2 to navigate to privacy related screens, Capture of actual network traffic and URL visits, Comparison of observed behavior with privacy policy disclosures

7.2 APK Collection and Dataset Preparation

7.2.1 Google Play Store Collection

Healthcare applications were collected from the Google Play Store using the `collect_top_chart.py` script, which leverages the `gplay-scraper` library to programmatically retrieve top-ranked apps in the Health & Fitness category.

Collection methodology:

Target category: `HEALTH_AND_FITNESS` (Google Play category identifier), **Ranking**:

TOP_FREE applications that are most downloaded, **Geographically**: European countries include Italy, Germany, France, Spain, the Netherlands, Luxembourg, Austria, Portugal, Ireland, and Finland, **Sample size**: Top 100 applications in each nation to adjust with the count argument, **Fallback strategy**: If primary country API fails, sequentially try alternative EU countries

Script usage:

```
python collect_top_chart.py \
  --outdir data/topcharts \
  --count 100
```

Output files:

- top_health_fitness_<country>_<date>.csv: Metadata including app ID, title, developer, genre, install count, rating, price, URL
- top_health_fitness_<country>_<date>.packages. txt: Plain text list of package names for batch download

The metadata CSV enables filtering by:

Install count (focus on widely-used apps with greater privacy impact), Developer (identify repeat offenders or privacy-conscious developers), Rating (correlate privacy practices with user satisfaction), Free vs. paid (analyze monetization strategies and third-party dependencies)

7.2.2 F-Droid Open Source Collection

To compare commercial Google Play apps with open-source alternatives, we collected apps from F-Droid, a repository of free and open-source Android applications, using download_fdroid_sports_health.py.

Collection methodology:

Download F-Droid repository index (index.xml) containing all available apps, Filter by category Sports & Health (F-Droid category taxonomy), For every matching application, download the latest APK version, Implement random delays of 0.8 to 2.5 seconds to avoid limiting

Script usage:

```
python download_fdroid_sports_health.py
# Output directory:  fdroid_sports_health_apks/
```

Rationale for F-Droid inclusion:

Transparency: Open-source apps should have minimal third-party dependencies and better privacy practices (hypothesis to test), **Baseline comparison**: F-Droid apps serve

as "privacy-conscious" baseline against which to compare commercial Play Store apps, **Reproducibility**: F-Droid apps are built from source with reproducible builds, ensuring the analyzed APK matches the published source
F-Droid collection challenges:

Smaller catalog: F-Droid has fewer health apps than Play Store (50 vs. 5000), **Lower install counts**: Less representative of typical user exposure, **Outdated apps**: Some F-Droid apps are unmaintained or deprecated

7.2.3 Dataset Filtering and Preparation

The collected APKs undergo preprocessing prior to analysis:

Deduplication: Remove duplicate package names, keeping highest version code, **Split APK handling**: Exclude configuration APKs
(e.g., `config.arm64_v8a.apk`, `split_config.xxhdpi.apk`) using regex patterns, **Validity check**: Ensure APKs are not corrupted using aapt dump badging test, **Size filtering**: Optionally exclude very small (100KB, likely stubs) or large (200MB, likely games misclassified as health apps) APKs

The `run_pipeline.sh` script implements smart APK selection for directories containing multiple APKs (e.g., Raccoon-downloaded bundles):

```
# If directory contains multiple base APKs:
# - Select APK with highest version code suffix (-<digits>. apk)
# - Example: app-123.apk preferred over app-120.apk
```

This ensures analysis targets the most recent version of each app while avoiding redundant processing of split APKs.

7.3 Static Analysis Phase 1: Taint Analysis with FlowDroid

7.3.1 Automated Source Generation

Health-specific taint sources are automatically generated by `generate_sources_from_apk.py`, which decompiles the APK and applies heuristics to identify methods likely to return health data.

Source identification heuristics:

Keyword matching: Class or method names containing health-related keywords (Table 7.1), **Function name patterns**: Methods starting with `get`, `load`, `read`, `fetch`, `calculate`, `compute` (typical accessor patterns), **Return type filtering**: Exclude `void` methods (setters) and focus on methods returning numeric, string, or object types, **Framework exclusion**: Skip Android/Google framework classes (`android.*`, `androidx.*`, `com. google. android.*`) to avoid noise

Table 7.1: Health data keywords for source detection

Category	Keywords
Weight	weight, weigh, bodyweight, bmi, bodymass
Body composition	bodyfat, body_fat, fat_pct, leanmass, lean_mass
Cardiovascular	heartrate, heart_rate, hr_bpm, hrv, pulse, rrinterval
Activity	steps, stepcount, step_count, walk_distance, run_distance
Energy	calories, kcal, cal_burned, energy_expended
Respiratory	spo2, sp_o2, oxygen_saturation, vo2max
Sleep	sleep_stage, sleep_quality, sleep_duration
Training	workout, training_session, training_load

Example generated source:

```
% from:  com/health/tracker/data/HeartRateRecord.java
<com.health.tracker.data.HeartRateRecord:  float getHeartRate()> -> _SOURCE_
```

Precision vs. recall trade off

The automated source identification strategy must balance two competing objectives in detecting health data access points: **high recall (liberal matching)** captures all potential health data sources through broad keyword matching and API pattern recognition, maximizing detection coverage but risking false positives where non-sensitive data is incorrectly classified as health-related; and **high precision (strict matching)** minimizes false positives through conservative filtering and strict type checking, ensuring that identified sources genuinely represent health data collection but risking false negatives where novel or obfuscated health data representations are missed.

Our approach favors **high recall** based on two methodological considerations. First, false positives (flows involving non-sensitive data incorrectly flagged as health-related) can be identified and filtered during post-processing by examining actual data semantics, parameter values, and execution context captured in FlowDroid's flow reports, allowing manual or automated review to eliminate spurious detections without re-executing expensive taint analysis. Second, false negatives (genuine health data flows missed during source identification) cannot be recovered in subsequent pipeline stages and represent permanent analysis blind spots that compromise study validity, as undetected privacy violations remain invisible in final results regardless of post-processing sophistication. This design decision prioritizes completeness over precision in initial detection, enabling comprehensive privacy violation discovery while deferring classification refinement to later stages where richer contextual information is available. In evaluation, we validate a random sample of auto-generated sources by manual code review to quantify the false positive rate.

7.3.2 Automated Sink Generation

Third-party library sinks are automatically identified by `generate_sinks_from_apk.py`, which detects embedded libraries and targets their data transmission methods.

Library detection pipeline:

Run android_lib_detector: Open-source tool that fingerprints libraries via package structure patterns, API call signatures, and string constants, **Extract package prefixes**: Parse `libs.csv` output to obtain root package (e.g., com.google.firebase, com.facebook.sdk), **Match decompiled classes**: Identify all classes whose fully qualified names start with library prefixes, **Filter by method names**: Within library classes, select methods with suspicious names.

Table 7.2: Method name substrings indicating third-party sinks

Category	Substrings
Analytics	track, event, log, record
User identification	identify, setuserid, setexternaluserid, setcustomeruserid
Properties	setuserproperty, setcustomuserattribute, tag
Transmission	send, push

Example generated sink:

```
% from: com/google/firebase/analytics/FirebaseAnalytics.java
<com.google.firebase.analytics. FirebaseAnalytics: void
logEvent(java.lang.String,android.os.Bundle)> -> _SINK_
```

Handling obfuscation:

Many third party libraries apply ProGuard or R8 obfuscation, renaming classes and methods to meaningless identifiers such as `a.b.c.d.e()`, rendering traditional package-name-based detection ineffective. The `android_lib_detector` tool addresses this through three complementary fingerprinting strategies. **Package structure fingerprints** leverage characteristic nested package hierarchies that persist after obfuscation, as obfuscators preserve nesting depth for bytecode compatibility. **String constant analysis** exploits hardcoded literals including API endpoint URLs (e.g., `https://graph.facebook.com/`), cryptographic keys, and version identifiers that survive obfuscation because runtime functionality depends on exact values. **API call patterns** identify libraries through sequences of Android API invocations constituting unique behavioral signatures detectable through control flow analysis despite method name obfuscation. These combined approaches enable reliable library detection in heavily obfuscated applications.

7.3.3 Combined Sources and Sinks Configuration

The `generate_sinks_from_apk.py` script merges three components into a unified `SourcesAndSinks_app.txt` configuration file for FlowDroid taint analysis. **SourcesAndSinks_base.txt** provides a manually curated baseline covering Health Connect,

Google Fit, Android sensors, and common tracking SDKs (Firebase, Facebook, AppsFlyer, Mixpanel). **auto_sources.txt** contains app-specific health data sources discovered through keyword-based scanning of decompiled code. **auto_sinks.txt** contains app-specific third-party library sinks identified through package analysis and SDK detection.

7.3.4 FlowDroid Execution

FlowDroid performs context-sensitive, flow-sensitive, object-sensitive taint analysis on the APK to detect data flows from sources to sinks.

Execution parameters:

```
java -Xms4g -Xmx64g -XX:+UseG1GC \
-jar flowdroid-2.14.1.jar \
-ne \                   # no-exceptions (ignore exceptional flows)
-a app.apk \
-p $ANDROID_HOME/platforms \
-s SourcesAndSinks_app.txt \
-o flowdroid_result.xml
```

7.3.5 FlowDroid Configuration and Timeout Management

Table 7.3 presents the FlowDroid configuration parameters and timeout handling strategy.

Table 7.3: FlowDroid Configuration and Timeout Management

Parameter/Aspect	**Configuration and Rationale**
Heap Size	64GB allocation for large healthcare apps with numerous third-party libraries requiring substantial memory for call graph construction
G1 Garbage Collector	Optimized for large heaps with predictable pause times, reducing analysis interruptions
No-Exceptions Mode (`-ne`)	Excludes flows through exception handlers, reducing false positives from error-handling code
Android Platforms	Required for resolving Android framework classes and system APIs
Timeout Duration	50-minute hard timeout via `gtimeout 3000` prevents indefinite execution on complex applications
Graceful Degradation	If timeout occurs but partial XML exists, post-processing proceeds with available flows
Status Tracking	Timeout events recorded in `taint_stats.csv` for complexity analysis. 6/110 apps (5.5%) exceeded timeout, all with ¿10 libraries

7.3.6 Flow Post Processing and Classification

Raw FlowDroid output (`flowdroid_result.xml`) contains low-level flow information requiring semantic interpretation. `postprocess_flows.py` enriches flows with human-readable classifications.

Post-processing pipeline:

Parse XML: Extract source method, source statement, sink method, sink statement for each detected flow, **Classify health category**: Match source method signature against health keyword patterns (Table 7.10), **Identify third-party library**: Match sink class against library prefixes from `libs.csv`, **Classify sink category**: Categorize sink as NETWORK, STORAGE, LOG, TRACKER, or THIRD_PARTY_OTHER based on class/method patterns, **Generate CSV**: Produce tabular output with columns: `flow_index`, `source_class`, `source_method`, `health_category`, `sink_class`, `sink_category`, `third_party_lib`, etc.

Table 7.4: Health data flow categories

Category	Representative Keywords
WEIGHT	weight, bmi, bodymass, leanmass
BODY_FAT	bodyfat, body_fat, fat, bf
HEART_RATE	heartrate, heart_rate, hr, pulse, hrv
STEPS	steps, stepcount, step_count
DISTANCE	distance
CALORIES	calories, kcal, energy_expended
VO2	vo2, vo2max
OXYGEN	oxygen, spo2
SLEEP	sleep
WORKOUT	workout, training
UNKNOWN	(no match)

Example classified flow (CSV output):

```
flow_index,source_method_sig,source_class,health_category,
sink_method_sig,sink_class,sink_category,third_party_lib
0,"<com.health.data.StepsRecord: long getSteps()>",
com.health.data.StepsRecord,STEPS,
"<com.google.firebase.analytics.FirebaseAnalytics: void
logEvent(java.lang.String,android.os.Bundle)>",
com.google.firebase.analytics.FirebaseAnalytics,
TRACKER,com.google.firebase.analytics
```

This example indicates: Steps data (`getSteps()`) flows to Firebase Analytics (`logEvent()`), classified as a TRACKER sink, with third-party library identified as Google Firebase.

Validation of classifications:

We validate health category classification accuracy by manually reviewing 200 randomly sampled flows:

- **True positives**: Flow correctly classified to appropriate health category (e.g., `getHeartRate()` → HEART_RATE)
- **False positives**: Flow misclassified (e.g., `getDeviceId()` misclassified as health data due to keyword collision)
- **Unknown (correct)**: Flow legitimately unclassifiable due to generic method names (e.g., `getValue()`) without context

Results: 92% accuracy, with false positives primarily from overly generic keywords (”data”, ”health”) matching non-health-specific methods.

7.4 Static Analysis Phase 2: Privacy Inspection

Complementing taint analysis, `health_privacy_static_analyzer.py` performs heuristic inspection for privacy indicators, third-party tracking SDKs, and security anti-patterns.

7.4.1 Privacy Policy and Terms Detection

Privacy Policy Detection Methodology

The privacy policy detection implements a multi-stage static analysis combining resource extraction, URL discovery, contextual classification, and status determination. The tool decompiles the APK using apktool for complete resource extraction including XML layouts, HTML assets, and configuration files. The decompiled directory is recursively scanned for text files (`.xml`, `.html`, `.txt`, `.json`, `.js`), and all HTTP/HTTPS URLs are extracted using the regex pattern `https?://[^\s"'<>\\)]+`.

Extracted URLs are classified through contextual keyword analysis examining a 120-character window around each URL. **Privacy URLs** have surrounding context containing *privacy*, *gdpr*, *informativa*, or *consent*. **Terms URLs** have context containing *terms*, *conditions*, *eula*, or *tos*. **Other URLs** have no contextual matches. Additionally, the tool searches for privacy keywords using regex `privacy(\s*policy)?|gdpr|data\s+protection`, recording matches with file paths and line numbers.

Applications are assigned three possible statuses: **YES** indicates at least one privacy-specific URL was found, suggesting explicit policy linkage. **MAYBE** indicates no privacy URLs but privacy keywords present, suggesting policy exists but not directly linked. **NO** indicates neither URLs nor keywords found, suggesting potential lack of privacy disclosure.

Challenges and limitations:

Obfuscation: String resources may be encrypted or base64-encoded, evading regex detection, **Dynamic loading**: Privacy URLs fetched from remote server at runtime are invisible to static analysis, **Language barriers**: Regex patterns optimized for English/Italian; apps in other languages may be missed, **False positives**: Generic URLs (e.g., `https://www.google.com/`) near privacy keywords may be misclassified

Despite limitations, evaluation on manually labeled ground truth (100 apps) shows 85% precision and 78% recall, sufficient for large-scale triage.

7.4.2 Third Party SDK Detection

Static analyzer detects presence of 14 common tracking/advertising SDKs by searching decompiled code for characteristic package prefixes and string constants (Table 7.5).

Table 7.5: Third-party SDK detection patterns

SDK	Package Prefix	String Hints
Firebase Analytics	com.google.firebase. analytics	google-analytics
Google Analytics (Legacy)	com.google.android. gms. analytics	google-analytics
Facebook SDK	com.facebook.appevents	graph. facebook.com
Mixpanel	com.mixpanel.android	api.mixpanel.com
AppsFlyer	com.appsflyer	appsflyer. com
Adjust	com.adjust.sdk	adjust.com
Segment	com.segment.analytics	api.segment.io
Branch	io.branch. referral	branch.io
Amplitude	com.amplitude.api	api.amplitude.com
Flurry	com.flurry.android	flurry.com
AdMob	com.google.android.gms.ads	googlesyndication
OneSignal	com.onesignal	onesignal.com
Kochava	com.kochava. base	kochava.com
Braze (Appboy)	com.appboy	appboy.com

Detection logic:

For each SDK: Check if any class in DEX file starts with package prefix → add SDK to detected list, Check if any string constant contains characteristic endpoint URL → add SDK to detected list. This dual approach (class-based + string-based) ensures detection even when SDKs are partially obfuscated or only partially integrated (e.g., library bundled but not used).

Risk assessment:

Detected SDKs are cross referenced with health permissions **High risk**: Tracking SDKs + health permissions (e.g., Firebase + `BODY_SENSORS`) → high likelihood of health data sharing, **Medium risk**: Tracking SDKs without health permissions → general behavioral tracking, no direct health data exposure, **Low risk**: No tracking SDKs

7.5 Dynamic Analysis: Runtime Behavior Validation

Dynamically loaded code (e.g., JavaScript injected into WebViews), Server side decisions (e.g., remote configuration determining which third-party SDKs to activate), Actual user-visible privacy disclosures (e.g., in-app privacy policy screens)

Dynamic analysis complements static analysis by observing apps in live execution environments.

7.5.1 Frida Based Network Interception

`frida_privacy_watch. js` is a Frida script (JavaScript) that hooks Android framework APIs to intercept URL accesses in real time.

Hooked API Methods for URL Monitoring

Table 7.6 presents the Frida-hooked API methods for monitoring URL loading and navigation.

Table 7.6: Frida Hooked API Methods

Category	**Hooked Methods and Purpose**
WebView URL Loading	`WebView.loadUrl(String)`, `WebView.loadUrl(String, Map)`, `WebView.loadDataWithBaseURL(...)`, `WebViewClient.shouldOverrideUrlLoading(...)`. Captures in-app web content loading
Intent-Based URL Opening	`Activity.startActivity(Intent)` with `ACTION_VIEW`, `ContextWrapper.startActivity(Intent)`, `Intent.setData(Uri)`. Captures URLs launched in external browsers
Custom Tabs	`androidx.browser.customtabs.CustomTabsIntent.launchUrl(...)`, `android.support.customtabs.CustomTabsIntent.launchUrl(...)`. Captures Chrome Custom Tabs usage

URL classification logic (JavaScript):

```
function classifyUrl(url) {
  const s = url.toLowerCase();
  if (s.includes("privacy") || s.includes("gdpr") ||
      s.includes("data-protection") || s.includes("informativa"))
    return "privacy";
  if (s.includes("terms") || s.includes("termini") ||
      s.includes("conditions") || s.includes("eula") ||
      s.includes("tos"))
    return "terms";
  if (s.includes("legal") || s.includes("cookie"))
    return "legal";
  return "other";
}
```

Event reporting:

When a hooked API is invoked, Frida sends message to Python controller:

```
send({
```

```
  type: "url",
  kind: "WebView. loadUrl(String)",
  category: "privacy",
  url: "https://example.com/privacy-policy",
  extra: ""
});
```

Python script aggregates events and classifies app as:

- **Privacy seen**: At least one URL classified as "privacy" was accessed
- **Terms seen**: At least one URL classified as "terms" was accessed
- **No privacy**: No privacy-related URLs observed

7.5.2 UI Automation for Privacy Policy Discovery

Many apps do not load privacy policies automatically on launch; users must navigate to Settings → About → Privacy Policy. Manual exploration is infeasible for hundreds of apps, so we automate UI navigation using **uiautomator2**.

Automation strategy: Keyword based clicking Search UI hierarchy for elements (buttons, menu items, text views) containing privacy related keywords: "privacy", "informativa", "gdpr", "settings", "about", "terms", etc. **Menu exploration**:

- Open overflow menu (three-dot icon, "More options" description)
- Open navigation drawer (hamburger icon, "Navigate up" description)
- Press the hardware menu button

Scrolling search: Scroll vertically through scrollable containers, repeatedly checking for privacy keywords, **Screen dumps**: Periodically dump XML UI hierarchy and regex-search for privacy keywords in `text` and `content-desc` attributes, **Activity launching**: For apps where UI exploration fails, directly launch Activities with privacy-related names (e.g., `com.app.PrivacyPolicyActivity`) via `adb shell am start`

7.6 Empirical Results

Analysis of 114 F-Droid health applications collected between December 2024 and February 2026 revealed significant privacy gaps.

7.6.1 Privacy Policy Disclosure

Table 7.7: Privacy policy disclosure status (N=114)

Status	Count	Percentage
YES (URL present)	63	55.3%
MAYBE (keywords only)	44	38.6%
NO (no indicators)	4	3.5%
Unclassified	3	2.6%

Only 55.3% provided explicit privacy policy URLs. The 38.6% "MAYBE" classification indicates privacy keywords without URLs, often representing incomplete implementations or third-party SDK references. Four applications (3.5%) showed no privacy indicators despite requesting sensitive permissions.

7.6.2 Third Party Library Prevalence

Table 7.8: Third-party tracking library detection (N=114)

Library	Apps	%
Firebase Analytics	28	24.6%
Crashlytics	31	27.2%
Google AdMob	15	13.2%
Facebook SDK	12	10.5%
AppsFlyer	4	3.5%
Mixpanel	3	2.6%
Any tracking library	**67**	**58.8%**

The 58.8% tracking library prevalence is lower than Google Play Store studies (80-90% [1, 12]), suggesting F-Droid's anti-features warnings provide partial deterrence.

7.6.3 Data Flow Analysis

FlowDroid analysis completed for 108 applications (94.7% success rate):

Critical Finding: Among 23 apps transmitting health data to analytics, 7 (30.4%) had inadequate privacy policies ("NO" or "MAYBE"), indicating undisclosed third-party sharing. This aligns with findings that third-party leaks are 5x more frequent than first-party leaks [1].

Table 7.9: Taint flow detection results (N=108)

Flow Category	Apps	%
Health to analytics	23	21.3%
Health to network	41	38.0%
Health to storage	89	82.4%
Health to advertising	8	7.4%
Any third-party flow	**52**	**48.1%**

7.6.4 Health Data Categories and Security Issues

Table 7.10: Health data categories in flows (391 total flows)

Category	Flows	%
Steps/Activity	127	32.5%
Weight/BMI	62	15.9%
Calories	53	13.6%
Heart Rate	48	12.3%
Distance	39	10.0%
Sleep	31	7.9%

Security anti-patterns detected: cleartext traffic allowed (27.2%), backup allowed without exclusions (80.7%), hardcoded API keys (23.7%), exported components without permissions (15.8%).

Chapter 8

Conclusions and Future Works

This chapter summarizes the contributions of this thesis, presents empirical findings from analyzing 114 F-Droid health applications, discusses implications for mobile health privacy, and proposes future research directions.

8.1 Summary of Contributions

This thesis developed an automated framework for detecting third party data sharing in Android health applications through static taint analysis combined with privacy policy inspection. The framework addresses the critical gap of systematically detecting undisclosed health data flows to third party advertising, analytics, and tracking libraries.

8.1.1 Key Contributions

Technical Contributions: Automated Health Source Generation: Keyword-driven decompilation analysis automatically identifying 86 health-related keywords across 9 categories from application bytecode, **Third-Party Sink Detection**: Library fingerprinting integrated with pattern matching to identify tracking sinks in analytics SDKs (Firebase, Facebook, Mixpanel, Adjust, AppsFlyer), **Semantic Flow Classification**: Post-processing framework enriching taint analysis with health categories, sink types, and third-party attribution, **Hybrid Validation**: Dynamic analysis using Frida instrumentation and UI automation for privacy policy verification

Methodological Contributions:

- Scalable pipeline processing 114 F-Droid applications with 95% completion rate
- Open-source implementation enabling independent replication
- Ground truth dataset for evaluating future privacy analysis techniques

8.2 Implications and Context

Our finding that 48.1% of F-Droid health applications exhibit third party data flows aligns with recent large-scale studies. Ardalani et al. [1] found third-party leaks 5x more frequent than first party leaks in 1,000+ health apps. Chen et al. [12] documented that 87% of fitness app transmissions involved cross app tracking identifiers. Hao et al. [2] found 72% of healthcare apps transmit sensitive data to third party libraries without consent. Our FDroid dataset shows lower prevalence (58.8% vs. 80-90% in Play Store), suggesting platform policy impact.

Privacy policy transparency challenges mirror broader literature findings. Wang et al. [18] found 63% of mHealth policies use vague third-party sharing language. Parker et al. [27] reported 87% of women's health apps share data with third parties, yet 13% provide no privacy information. Martinez et al. [16] documented pre consent data transmission "dark patterns."

Regulatory Concerns. Detection of health data flows to analytics without disclosure raises GDPR Article 13(1)(e) violations (third-party recipient disclosure) and Article 9 consent requirements for special category data [11, 10]. Recent FTC/HHS enforcement against Meta Pixel and Google Analytics tracking resulted in $100M+ settlements [20, 15, 28].

Industry Changes. Our findings occur during major policy shifts: Meta's 2024-2025 healthcare data restrictions [14, 22], CMS interoperability initiatives [25, 26], and evolving SDK privacy defaults [4, 29]. User studies show third-party sharing is the #1 adoption barrier for mHealth apps [19, 13].

8.3 Limitations

Static Analysis: FlowDroid cannot detect reflection based flows, runtime class loading, or server side behavior. Encrypted payloads prevent content inspection.

False Positives: Health keywords like "weight" appear in non health contexts. Manual validation (n=50) identified 12% false positive rate. Dead code and debug only logging may inflate results.

Dataset Bias: FDroid's privacyconscious selection underestimates Play Store prevalence. Italian locale settings may miss non-Italian policies. Dataset represents December 2024 February 2026 versions only.

Validation Scope: Manual validation covered 12.8% of flows. Dynamic analysis was executed on only 31 apps due to infrastructure constraints.

8.4 Future Work

Enhanced Detection: Hybrid static dynamic analysis combining FlowDroid with runtime taint tracking [17] could reduce false positives and detect reflection-based flows. Machine learning classifiers trained on validated samples could improve source identifica-

tion beyond keyword matching. Network traffic analysis via SSL/TLS interception could reveal actual payload contents.

Longitudinal Studies: Version evolution tracking could quantify privacy trends as apps mature [29]. Regulatory impact assessment comparing pre/post GDPR or pre/post-Meta restrictions could measure policy effectiveness. Platform comparison (Play Store vs. FDroid builds) could isolate platform policy influence.

Privacy Policy Automation: NLP based policy code consistency verification could detect contradictions between stated policies and actual flows [18]. Automated third party processor identification from policy text could flag undisclosed relationships. Dynamic consent mechanism validation could detect pre consent data transmission [16].

Developer Tools: Privacy linting as Android Studio plugin could provide real time feedback. Automated privacy policy generation from detected flows could reduce developer burden. SDK privacy configuration templates could enable privacy by design [30].

Broader Threat Modeling: Cross app tracking analysis examining advertising ID linkage across apps. Data broker flow analysis extending sink detection to commercial data markets. Inference attack modeling assessing privacy risks from behavioral metadata [31, 32].

8.5 Closing Remarks

This thesis demonstrates that automated static analysis can systematically detect third party health data sharing at scale. Analysis of 114 FDroid applications revealed 48.1% transmit health data to third party libraries, with 30.4% lacking adequate privacy disclosure evidence that mobile health privacy depends critically on third party flows, not just first party collection.

The results add to the mounting evidence of privacy practices that deviate from legal standards and user expectations. Increased regulatory attention is reflected in recent enforcement against healthcare tracking pixels [15, 20] and changes in industry policy [14]. But technological detection is not enough on its own. Regulation, platform policy, SDK vendors, and developer tools must all work together to address ecosystem issues like default opt out SDK configurations [4], ambiguous policy language [18], and pre consent transmission [16].

Because health apps include third-party SDKs whose data gathering methods developers may not entirely control, health app privacy is really a *supply chain* issue. Testing SDKs for data collecting behavior is necessary for developing privacy preserving health apps, much as testing dependencies for security flaws is necessary for secure development. Sustainable privacy protection necessitates moving from reactive detection to proactive prevention through privacy preserving SDK defaults, platform level restrictions (like iOS App Tracking Transparency), and regulatory incentives that balance vendor and user interests. The framework created here offers one part of that vetting process.

Bibliography

[1] Niousha Ardalani, Yiming Chen, and Lei Wang. Towards precise detection of personal information leaks in mobile healthcare applications. *arXiv preprint arXiv:2410.00277*, 2024. Analyzes PI leaks to third parties in 1000+ health apps; finds third-party leaks 5x more frequent than first-party.

[2] Zhihui Hao, Xiaohui Liu, and Ming Wang. Detection and privacy leakage analysis of third-party libraries in android healthcare apps. *Proceedings of the International Conference on Information Security*, pages 1–15, 2024. Introduces Libmonitor tool for detecting third-party library leaks in healthcare apps.

[3] Michael Zhou. An empirical study of privacy leakage vulnerability in third-party android logging libraries. Master's thesis, University of Waterloo, 2023. Analysis of 50,000 apps; finds logging libraries leak PI in healthcare apps.

[4] M. Rodriguez, Y. Chen, and H. Wang. Privacy settings of third-party libraries in android apps: A study of facebook sdks in healthcare. *NSF Public Access Repository*, 2023. Examines default privacy settings in Facebook SDK used by health apps.

[5] Statista. Android top mobile app analytics sdks 2025, 2025. Statistical analysis of leading mobile app analytics SDKs on Android platform.

[6] 42matters. All categories for google play sdk stats, 2024. Comprehensive SDK analysis and statistics for Google Play applications.

[7] Appfigures. Top analytics sdks installed in ios & android apps and games, 2024. Market analysis of top analytics SDKs across mobile platforms.

[8] Mobio Group. Android and ios sdks: The leaders of 2024. *Mobio Group*, 2024. Industry report on leading mobile SDKs for Android and iOS platforms.

[9] AppBrain. Statistics of android apps, 2024. Comprehensive statistics and market data for Android applications.

[10] Legal Information Foundation. Hipaa and gdpr compliance for health app developers: Third-party data sharing requirements. *LLIF. org*, 2025. Specific focus on third-party business associate agreements and data sharing.

[11] MoldStud Development. Gdpr compliance for healthcare apps: Managing third-party data processors. *MoldStud Articles*, 2024. Step-by-step guide for GDPR Article 28 (third-party processors).

[12] Y. Chen, L. Wang, and T. Zhang. A study of personal information leaks in mobile fitness apps. In *International Journal of Web Information Systems*, pages 207–225, 2024. Documents third-party data sharing in fitness/health apps.

[13] Office of the National Coordinator for Health IT. Individuals' access and use of patient portals and smartphone health apps: Third-party aggregators analysis. *HealthIT.gov Data Brief*, 2024. Only 7% use third-party aggregation apps due to privacy concerns.

[14] Cardinal Digital Marketing. How meta's data restrictions impact healthcare advertising strategies. *Cardinal Digital Marketing Blog*, 2024. Analysis of Meta's 2025 restrictions on healthcare data sharing with advertisers.

[15] Federal Trade Commission and Department of Health and Human Services. Ftc and hhs warn hospital systems and telehealth providers about privacy and security risks from online tracking technologies. *FTC Press Release*, July 2023. Official regulatory warning about Meta Pixel and Google Analytics in healthcare.

[16] A. Martinez, B. Schmidt, and C. Johnson. Transparency and consent challenges in mhealth apps: An empirical study of third-party data flows. In *Proceedings of the International Conference on Privacy and Security*, pages 1–18. Springer, 2023. Dark patterns and pre-consent data transmission to third parties.

[17] R. Patel and S. Kumar. Comprehensive survey on privacy analysis of android applications: Leak detection with focus on third-party libraries. *Journal of Emerging Technologies and Innovative Research*, 11(9), 2024. Survey of leak detection methods for third-party libraries in Android health apps.

[18] Haoyu Wang, Xiao Liu, and Yuqing Zhang. A qualitative analysis framework for mhealth privacy practices. *arXiv preprint arXiv:2405.17971*, 2024. Framework for analyzing privacy policies and third-party sharing in mHealth apps.

[19] Niousha Mehrabi, Fred Morstatter, and Nripsuta Saxena. Listening to users: Privacy and security in mobile health apps. In *Proceedings of the International Conference on Human-Computer Interaction*, pages 45–62. Springer, 2024. User perception study of privacy in mHealth apps and third-party sharing.

[20] Feroot Security. Pixel tracking violations cost us healthcare over $100 million. *Feroot Security Blog*, 2024. Analysis of HIPAA violations from third-party tracking pixels (Meta, Google) in healthcare apps and websites.

[21] Mintz Legal Counsel. Ocr and ftc issue joint statement warning health care providers and app developers. *Mintz Insights*, July 2023. Legal analysis of regulatory enforcement on third-party tracking.

[22] All Health Tech. Navigating meta's 2025 data restrictions: A new era for healthcare privacy. *All Health Tech*, 2025. Impact analysis of advertising network restrictions on healthcare data.

[23] Accelerated Digital Media. Meta's new data sharing policy could change health marketing in 2025: Here's what to know. *Accelerated Digital Media Insights*, 2025. Healthcare marketing implications of third-party data sharing restrictions.

[24] Xiaohui Li, Jing Wang, and Wei Zhang. A trusted medical data sharing framework for edge computing leveraging consortium blockchain. *Heliyon*, 9(11):e21342, 2023. Proposes blockchain solution to control third-party access to medical data.

[25] MedCity News. 60+ companies join cms' new initiative for data interoperability: Balancing sharing and privacy. *MedCity News*, July 2025. CMS initiative with Apple, Google, Amazon addressing secure third-party data sharing.

[26] STAT News. Health care, tech companies promise to make patient data more accessible while protecting from third parties. *STAT News*, July 2025. Analysis of industry pledges to limit unauthorized third-party access.

[27] L. Parker, M. Johnson, and S. Williams. Privacy, data sharing, and data security policies of women's mhealth apps. *Women's Health Technology*, 2024. 87% of women's health apps share data with third parties; 13% provide no information.

[28] ByteBack Law. Healthcare website tracking: Lessons from four recent ecpa rulings. *ByteBack Law Blog*, September 2025. Legal analysis of Electronic Communications Privacy Act cases involving healthcare tracking.

[29] C. Ren, P. Liu, and K. Chen. Big fixes, improvements, and privacy leaks: A longitudinal study of pii leaks across android app versions. *FTC PrivacyCon*, 2023. Longitudinal analysis showing increasing third-party data collection in health apps over time.

[30] Help Net Security Editorial Team. The diagnosis is in: Mobile health apps are bad for your privacy. *Help Net Security*, October 2024. Investigation revealing widespread third-party data sharing in mHealth apps.

[31] NowSecure Research Team. Top mobile app security and privacy breaches of 2024: Third-party data sharing in healthcare. *NowSecure Blog*, January 2024. Documents major healthcare app breaches involving third-party data sharing.

[32] Data Insights Market. Health app risks: Privacy and efficacy concerns in 2025 - third-party data sharing edition. *Data Insights Market*, 2025. Market analysis of third-party data sharing risks in health apps.