An Empirical Study of Privacy Leakage Vulnerability in Third-Party Android Logs Libraries

by

Yixi Zhao

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2025

© Yixi Zhao 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Mobile logging libraries are essential tools for debugging and monitoring Android applications, yet their privacy implications remain largely unexplored. This paper presents the first large-scale empirical study of privacy risks in Android logging practices, analyzing 48,702 applications from Google Play to identify sensitive data leakage through third-party logging frameworks. Our findings reveal that while logging library adoption is limited (3.4% of applications), nearly half (49.3%) of logging-enabled applications exhibit privacy leaks, creating significant security vulnerabilities. Three dominant libraries—Timber (35.2%), SLF4J (35.1%), and Firebase (29.4%)—account for 99.7% of all verified privacy leakage instances.

We identify distinct logging patterns across frameworks, with SLF4J showing balanced log level distribution, Timber concentrating heavily on DEBUG levels (78.5%), and Firebase dominated by Analytics Events (98.0%). Our analysis reveals that privacy violations predominantly stem from indirect data flows (62.5%) requiring intermediate processing steps, with most leaks occurring through moderate-complexity paths of 2-4 statements. User-info sources dominate privacy leaks (69.7%), while user-input sources represent a substantial portion (30.3%), highlighting GUI components as significant risk vectors.

Longitudinal analysis of application updates demonstrates that privacy leaks tend to improve over time, indicating growing developer awareness of privacy concerns, though persistent vulnerabilities underscore the need for systematic privacy protection measures. Our study contributes the largest dataset of third-party logging-based privacy violations to date, a reproducible analysis pipeline for future research, and actionable insights for developers and library maintainers. These findings emphasize the critical need for practitioners to recognize both user information and user input as significant privacy threats when implementing third-party logging frameworks in Android applications.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Weiyi Shang, for welcoming me into his research group and providing unwavering guidance throughout my master's studies. His mentorship has been invaluable in shaping my academic development.

I extend my sincere appreciation to Dr. Kundi Yao, whose expertise as a postdoctoral researcher has been crucial to the success of this work. His insightful contributions, technical guidance, and dedication to addressing complex research challenges have greatly enhanced the quality and depth of my research.

I am grateful to Professor Meiyappan Nagappan and Professor Jialu Zhang for serving as my thesis committee members. Their thorough review, constructive feedback, and valuable insights have been instrumental in refining and strengthening this thesis.

I would like to thank my fellow researchers in the Software Engineering and System Engineering (SENSE) research group for creating a collaborative and supportive environment. Their friendship, intellectual discussions, and shared experiences over the past years have made this journey both enriching and enjoyable.

Finally, I dedicate this work to my family and friends, whose love, encouragement, and unwavering support have been the foundation that made this achievement possible.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures viii

List of Tables ix

1 Introduction 1

2 Background 4

1 Flowdroid and taint analytics 4

2 Third part logging library 5

3 Case Study Setup 6

1 Approach 6

1.1 Dataset collection 7

1.2 Sink and source specification 7

1.3 Taint analysis pipeline 10

1.4 Empirical Analyses 11

4 Case Study Result **13**

1 RQ1: Which third-party logging libraries are most associated with privacy leaks? . . . 13

1.1 Motivation . . . 13

1.2 Approach . . . 14

1.3 Results . . . 14

2 RQ2: Which logging levels and data sources most frequently expose sensitive data? . . . 18

2.1 Motivation . . . 18

2.2 Approach . . . 19

2.3 Results . . . 19

3 RQ3: How complex are the data-flow paths between sensitive data sources and logging statements? . . . 23

3.1 Motivation . . . 23

3.2 Approach . . . 24

3.3 Results . . . 26

4 RQ4: How do privacy leaks in third-party logging libraries evolve across application updates? . . . 29

4.1 Motivation . . . 29

4.2 Approach . . . 29

4.3 Results . . . 30

5 Threats to Validity **31**

1 Internal Validity . . . 31

2 External Validity . . . 32

3 Construct Validity . . . 32

6 Related Work **33**

1 FlowDroid Analytics and Mobile Log Analytics . . . 33

2 User Input Threats to User Privacy . . . 34

7 Conclusion 35

References 37

List of Figures

2.1 Overview of Flowdroid workflow 5

3.1 Overview of our research worklow. 6

4.1 Distribution of logging libraries across whole vulnerabilities. 16
4.2 Overview of Leaking records. 17
4.3 Overview of log libraries across Valid leaking records. 17
4.4 SLF4J Log Levels. 20
4.5 TIMBER Log Levels. 21
4.6 Firebase Log Levels. 22
4.7 Leaking Source Type Distribution 22

List of Tables

3.1 Summary of Sink and Source libraries used in FlowDroid analytics. 9
3.2 Sample Source Methods for Sensitive Data Collection 10

4.1 Overview of FlowDroid Analysis Outcome 15
4.2 Summary of Log Statements and Source Connection Type 27
4.3 Summary of Data Flow path Length 28
4.4 Overview of Applications with Leaking Records Comparing with 2023 Version 30

Chapter 1

Introduction

Mobile applications have become an integral part of daily life, serving as essential tools for communication, entertainment, and productivity. Such applications handle vast amounts of sensitive user information ranging from personal communications to authentication credentials. The Android ecosystem, dominated by the Google Play Store, serves billions of users worldwide, making privacy protection in mobile applications a critical concern for both researchers and practitioners. While Google Play Store mandates that developers provide privacy policies and comply with relevant privacy regulations [12], significant privacy violations persist throughout the ecosystem. Empirical studies reveal a persistent gap between declared practices and actual behavior, with Guamán et al. [14] finding that 66% of popular applications contain ambiguous, inconsistent, or other compliance issues when analyzed against privacy regulations.

Among the various mechanisms that can lead to such data exposure, logging represents a particularly concerning yet understudied vector. Logging is a central engineering practice used to diagnose errors, monitor systems, and track processes [17, 15, 23], and is especially valuable in mobile development where many faults surface only on user devices in production environments. Zeng et al. [27] analyzed applications from F-droid and concluded that the majority of mobile logging occurs at debug and error levels and is frequently modified during development. Additionally, Zhou et al. [30] found that mobile log statements are widely used in Activity components, representing a significant portion of logging activity within Android applications. While mobile logging serves as an important development tool, it also poses a threat to user privacy. Existing research [30, 10] has demonstrated that Android logs, as important resources, contain privacy leakage vulnerabilities.

The privacy risks associated with logging are significantly amplified by the widespread

adoption of third-party logging frameworks in modern Android development. Unlike Android's built-in logging utilities, third-party logging libraries such as Timber, SLF4J, and Firebase offer enhanced functionality, flexible formatting options, and integration with remote monitoring platforms [16]. These libraries have become increasingly popular among developers seeking more sophisticated logging capabilities beyond the basic Android Log API. However, their enhanced features introduce additional privacy risks: they may buffer log data locally, transmit logs to remote servers for analytics, or persist sensitive information longer than intended. Studies have shown that third-party libraries can contain privacy violations leading to sensitive data leakage [8, 22], and logging frameworks represent a particularly concerning category due to their explicit role in data collection and potential for remote transmission.

Despite these elevated risks, existing privacy research has largely overlooked third-party logging libraries, representing a critical research gap in mobile privacy analysis. Privacy leakage detection in Android applications has been extensively studied, with researchers primarily focusing on unauthorized data transmission through network communications and third-party libraries [31, 6, 29, 21, 24]. Static analysis tools like FlowDroid [5] have proven effective in detecting privacy violations by tracking sensitive data flows from sources to potential leak points. However, most existing work either focuses on Android's native logging API or considers logging only as a minor category of data sinks, with comprehensive investigation of logging-specific privacy risks across third-party frameworks remaining limited, particularly at the scale necessary to understand their real-world impact. Furthermore, existing privacy analysis frameworks have primarily focused on Android API methods from sources like the SUSI framework [6] while neglecting user input through graphical user interfaces, which represents a significant source of sensitive information that could be inadvertently captured and exposed through logging statements.

To address this research gap, this thesis presents a systematic empirical study of logging-related privacy leaks with particular emphasis on third-party logging frameworks. We build a custom taint analysis pipeline on top of FlowDroid [5], using an extended source list that includes both system-provided sensitive APIs from SUSI [6] and GUI-based user-input methods, and a curated sink list covering ten popular third-party logging libraries. Applying this pipeline to 48,702 Google Play applications spanning from 2016 to 2021, we produce a validated dataset of confirmed privacy leaks through comprehensive filtering and manual validation processes. Using this dataset, we investigate four research questions to provide a comprehensive view of the privacy risks posed by logging in real-world Android applications: (1) the prevalence of logging-based leaks and which third-party libraries are most implicated, (2) the relationship between logging severity levels and the categories of sensitive data being leaked, (3) the structural complexity of data-flow paths from sensitive

sources to logging sinks, and (4) whether these leaks persist, disappear, or worsen in updated versions of the same applications over time.

Our study contributes three primary outcomes.

- We provide a large-scale measurement of logging-related privacy leaks focused explicitly on third-party frameworks, offering empirical evidence about the scope and concentration of risk across popular libraries.
- We deliver a documented analysis pipeline that combines automated FlowDroid runs, deterministic preprocessing and filtering, and targeted manual validation to yield a high-precision leak dataset suitable for reproducible research.
- We report actionable empirical insights and discuss implications for developers, library maintainers, and tooling designers. These insights include which libraries and logging practices are most associated with leaks, the typical structural patterns by which sensitive data reaches log sinks, and whether these issues persist in more recent app releases.

Thesis organization. The thesis is structured as follows: Chapter 2 introduces the background in Flowdroid and taint analytics, Third part logging library. Chapter 3 describes the subjects we have studied and how we extract the related data. Chapter 4 explains the motivation, approach, and results of each research question. Chapter 5 presents the threats to validity of the study. Chapter 6 discusses the related work. Finally, Chapter 7 concludes this thesis.

Chapter 2

Background

In this chapter, we will introduce the Flowdroid and taint analytics and the third party logging libraries.

1 Flowdroid and taint analytics

Taint analysis is a security analysis technique that traces untrusted data from its entry point (the source) to sensitive operations (the sink) to identify potential security vulnerabilities. In Java applications, taint analysis can precisely identify data flow paths through context-sensitive pointer alias analysis [19]. Considering the lifecycle differences between Android and traditional Java programs, we cannot directly apply standard taint analysis tools to Android applications. FlowDroid provides a precise model targeting Android's unique lifecycle, allowing the analysis to properly handle callbacks invoked by the Android framework [5, 7].

Figure 3.1 illustrates the general taint analysis workflow using FlowDroid according to previous research [5, 29, 28]. The FlowDroid taint analysis begins with downloading the APK dataset, followed by defining the requirements for sink and source methods. Subsequently, we configure FlowDroid parameters according to our research objectives. After implementing FlowDroid analysis across the APK dataset, we conduct parallel evaluation of the leaking reports and intermediate Jimple code for specific cases. Finally, we integrate the analysis results to generate comprehensive findings.

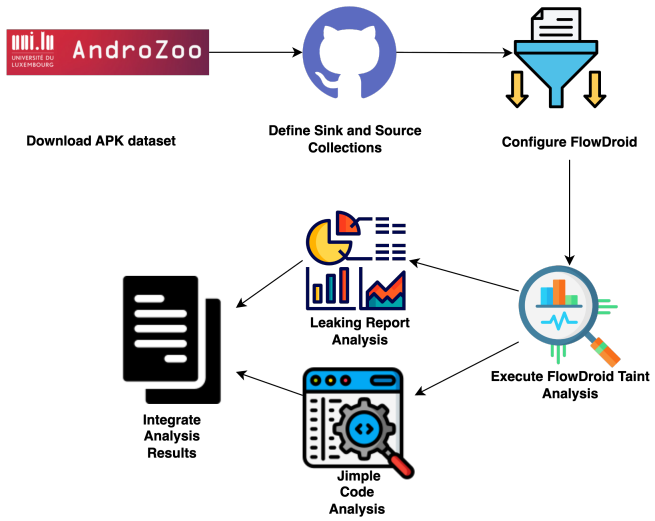


Figure 2.1: Overview of Flowdroid workflow

2 Third part logging library

In the Android system, logging libraries can be categorized into two types: official logging mechanisms that are embedded within the official SDK, and third-party logging libraries developed by external organizations or developers.

Official Android Logging: The Android SDK [1] provides built-in logging functionality through the android.util.Log class, which offers standard logging levels (VERBOSE, DEBUG, INFO, WARN, ERROR) and is directly integrated into the Android framework.This official logging system is designed primarily for development and debugging purposes, with logs typically stored locally on the device.

Third-Party Logging Libraries: Third-party logging libraries are not built into the Android SDK directly; they are external frameworks that extend or replace Android's native logging capabilities. Popular third-party logging libraries include SLF4J, Timber, Logback, and Firebase Analytics. These libraries often provide enhanced features such as remote logging, advanced filtering, structured logging, and integration with analytics platforms. However, these libraries may transmit log data to remote servers or process information in ways that differ from the standard Android logging behavior, potentially introducing privacy risks that are not present in native Android logging mechanisms.

[1] https://developer.android.com/reference/android/util/Log

Chapter 3

Case Study Setup

In this chapter, we explain the process of collecting sink and source API methods and applying them to the FlowDroid pattern, as well as how we implement taint analysis through FlowDroid and conduct quantitative and qualitative analysis.

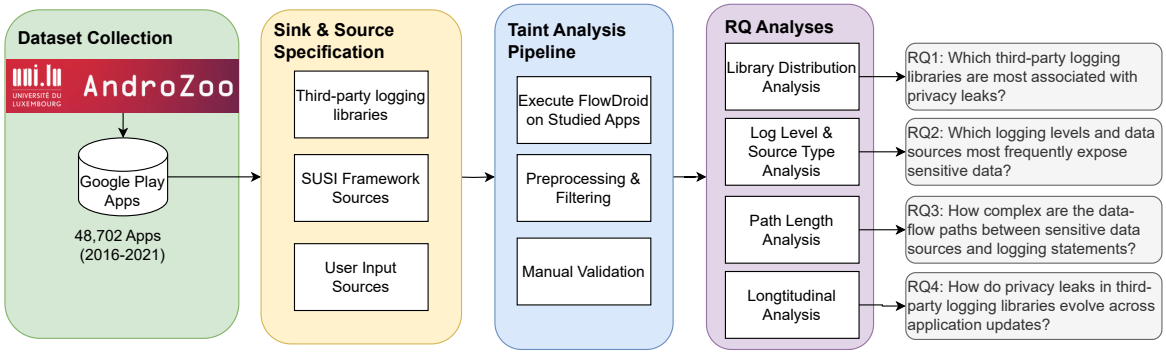


Figure 3.1: Overview of our research worklow.

1 Approach

Our approach consists of four stages, shown in 3.1: dataset collection, sink and source specification, static taint analysis, and RQ-specific analyses. This pipeline ensures reproducibility and aligns with our research questions (RQ1–RQ4) by progressively refining data and generating structured outputs at each stage.

1.1 Dataset collection

For the subjects under study, we utilized the AndroZoo dataset. AndroZoo[1] is a comprehensive repository containing millions of Android applications collected from various sources including Google Play Store, F-Droid, and other alternative markets [2]. This dataset has been extensively used in prior studies for malware detection, privacy analysis, and large-scale Android application research [3, 18, 1].

While AndroZoo contains applications from multiple sources, we strategically focused our analysis on applications exclusively from Google Play Store for several reasons. First, Google Play Store represents the dominant Android application distribution platform, accounting for the vast majority of Android app installations worldwide, making it the most representative sample for understanding real-world privacy practices. Second, Google Play Store enforces stricter quality assurance and security screening processes compared to alternative markets, which means that privacy violations found in these vetted applications are particularly concerning and likely to affect a broader user base. Third, Google Play Store mandates that all developers provide privacy policies and comply with relevant privacy regulations [12], creating a controlled environment where we can examine the gap between stated privacy practices and actual application behavior. Finally, empirical studies demonstrate that more than 50% of applications contain privacy policies [25], providing sufficient ground truth for our privacy-focused analysis.

Through AndroZoo's RESTful API, we downloaded 48,702 Android applications from Google Play Store. The collected APK files span from 2016 to 2021, with the majority distributed between 2019 and 2021. To ensure the reliability of our analysis, we exclude the applications released after 2021, as FlowDroid exhibits compatibility issues with newer Android API versions due to evolving source/sink specifications and bytecode format changes [5].

1.2 Sink and source specification

In this step, we first identify the sink and source libraries, then extract the relevant API methods from these libraries, and finally format them for static analysis.

[1] https://androzoo.uni.lu/

Identify Sink and Source Libraries

Sink Libraries. Building upon prior research by [30], which demonstrated privacy vulnerabilities in Android's official logging utility (`android.util.Log`), our investigation focuses exclusively on third-party logging frameworks not integrated into the core Android SDK. Through manually analysis of GitHub [2] repositories and Android Arsenal [3], we identified 10 distinct third-party logging libraries commonly employed in Android applications, with selection criteria based on GitHub star ratings and comprehensive library documentation. Eight of the selected library repositories exceed 1,000 GitHub stars, indicating their popularity as logging frameworks in Android development. While Tinylog has 744 stars, it was included due to its mature documentation and extensive development history with 83 releases, indicating active maintenance and community adoption. Firebase logging methods were included as a separate category due to their unique event-based logging paradigm rather than traditional log levels.

Source Libraries. To enable FlowDroid to identify data flows from sensitive inputs to logging sinks, we constructed a comprehensive source specification combining both system- and user-level data sources. First, we adopted the SUSI framework [6], which provides a curated set of Android API methods that access sensitive system information. Second, to capture user-provided data, we extended this list with user-input related sources. We systematically collected methods from the official Android API documentation[4] that correspond to GUI event callbacks or input widgets (e.g., `getText()` on `EditText`). We also incorporate input-handling methods from popular third-party libraries that provide custom UI components or input collection functionality, through a similar approach to locate them in Android Arsenal by their popularity. The resulting combined source list ensures that our analysis captures a wide range of sensitive information, including both device-level attributes and dynamic user content.

A summary of the identified sink and source libraries is presented in Table 3.1.

Sink and Source Method Extraction and Formatting.

Sink Methods. We systematically extracted logging API methods from each of the 10 identified third-party logging libraries by analyzing their official documentation and source code repositories. This process yielded a total of 417 sink methods across all libraries, with

[2] https://github.com/
[3] https://web.archive.org/web/20250206183038/https://android-arsenal.com/
[4] https://developer.android.com/reference

Table 3.1: Summary of Sink and Source libraries used in FlowDroid analytics.

Type	Library	Stars	Methods	Main Purpose
Sink Libraries	Tinylog	744	90	Lightweight logging framework for Android
	XLog	3,200	84	Lightweight and flexible Android/Java logger
	Logback	3,100	61	Logging framework for Java
	Timber	10,600	42	Logger with small, extensible API
	Log4J	3,500	36	Industrial-grade Java logging framework
	KLog	1,900	33	Log tool for Android
	SLF4J	2,400	29	Simple Logging Facade for Java
	Logger	13,900	24	Simple, pretty and powerful Android logger
	Chucker	4,200	12	HTTP inspector for Android
	Firebase	N/A	6	Web application development platform
Source Libraries	SUSI	N/A	46	Android API source methods
	User input	N/A	158	User input related source methods

method counts ranging from 6 methods for Firebase to 91 methods for Tinylog, as detailed in Table 3.1. Each method was categorized according to its logging level (e.g., debug, info, warn, error) and functionality (e.g., standard logging, event logging, HTTP inspection).

Source Methods. We constructed our source collection from two primary components: 46 sensitive Android API methods from the SUSI framework [6] for accessing system information such as device identifiers and location data, and 158 manually collected Android GUI API methods from the Core Android Framework, Material Design components, and widely-used third-party input libraries. The user input collection comprises 130 methods (82.3%) derived from Android API methods and 28 methods (17.7%) across popular third-party libraries, covering various input types including text fields, search views, and keyboard events as illustrated in Table 3.2.

Formatting. Following data collection, we manually formatted all 621 API methods (417 sink methods and 204 source methods) according to FlowDroid's sink and source pattern specifications [5] to ensure compatibility with the static analysis engine. This formatting process involved converting method signatures to FlowDroid's required syntax, specifying parameter types, and defining appropriate access modifiers for each method entry.

Table 3.2: Sample Source Methods for Sensitive Data Collection

Method Type	Sample Method	Description
Edit Text	`android.text.Editable.getText()`	Retrieves text content from editable text fields where users input sensitive information
Text View	`android.text.Editable.getEditableText()`	Accesses editable text content from text display components that may contain user data
Search View	`java.lang.CharSequence.getQuery()`	Extracts search queries entered by users, potentially revealing search behavior and interests
Keyboard Event	`android.text.Editable.getEditable()`	Captures text input from keyboard interactions, including passwords and personal information
3rd Party Library	`java.lang.String.getCvv()`	Retrieves credit card verification codes from third-party payment processing libraries

1.3 Taint analysis pipeline

Next, we conduct taint analysis on the collected applications using FlowDroid [5], and preprocess the analysis results for subsequent empirical evaluation.

Flowroid Execution

To investigate data leakage issues in logging methods, we leveraged FlowDroid to perform taint analysis on applications downloaded from AndroZoo [2], using the sink and source collections generated in the previous step. We executed FlowDroid on the 48,702 collected APKs with a standard configuration: a 20-minute timeout to prevent infinite loops during analysis, an initial heap size of 1GB, and a maximum heap size of 4GB per thread. To optimize performance, we ran four parallel threads to analyze applications concurrently. During the analysis process, we recorded both FlowDroid leakage reports and runtime log files for comprehensive evaluation. For applications with confirmed leaks, we additionally extracted each APK's Jimple code representation to enable detailed analysis of the vulnerability patterns.

Preprocessing and Filtering FlowDroid Results

As raw FlowDroid outputs may contain noise and false positives, we implemented a multi-starge preprocessing pipeline to clean and filter the results to ensure analytical validity. First, we parsed FlowDroid reports into a unified schema, merging per-app results into a single structured dataset. Next, we applied deterministic filtering rules to remove records with irrelevant source methods such as generic `toString()` or `hashCode()` calls that could lead to false positives and spurious correlations. This data cleaning procedure was essential to maintain the integrity of our dataset and prevent erroneous associations between logging statements and unrelated code segments. The resulting preprocessed dataset enabled precise analysis of the semantic and structural relationships between logging statements and their originating source code, forming the foundation for subsequent empirical evaluation.

Manual validation of detected leaks

Finally, we performed a manual validation step to further improve precision. Ambiguous or high-impact cases were reviewed by examining the corresponding Jimple traces to confirm the existence of a valid data flow. To speed up this process, we used an LLM-assisted helper to summarize long traces and highlight the final sink calls, but all inclusion decisions were made by a human reviewer. More detailed methodology and examples of this trace inspection process are presented in RQ3.

The results of the taint analysis pipeline are then used for empirical analyses to answer our research questions.

1.4 Empirical Analyses

We conducted several empirical analyses based on the preprocessed FlowDroid results. Each analysis is tailored to address specific aspects of logging-related privacy leaks in Android applications.

Library Prevalence Analysis (RQ1)

To determine which third-party logging libraries are most associated with privacy leaks, we analyzed the distribution of each logging library across the validated leaking records. This analysis characterizes the prevalence of different logging frameworks in privacy-compromised applications and identifies which libraries pose the greatest privacy risks to users.

Logging Pattern Analysis (RQ2)

To investigate the relationship between logging levels and sensitive data sources, we examined two key characteristics for each detected leak: log levels (determined from library documentation) and source categories. We classified sources into two distinct types: user information sources (derived from SUSI framework [6] for system data like device identifiers) and user input sources (dynamic user-generated content from GUI interactions). This analysis reveals how different logging severities correlate with various types of sensitive information exposure.

Data Flow Complexity Analysis (RQ3)

To understand the execution path lengths between sensitive sources and logging sinks, we analyzed the Jimple intermediate representation files and categorized flows into direct connections (immediate source-to-sink paths) and indirect connections (paths with intermediate program statements). We systematically measured path lengths to evaluate data flow complexity patterns and characterize how sensitive information propagates through application code before reaching logging statements.

Temporal Evolution Analysis (RQ4)

To assess whether privacy leaks persist or improve over time, we conducted a longitudinal comparison using updated application versions. We collected the most recent versions (as of 2023) of applications that exhibited logging-based privacy leaks in our initial 2016-2021 dataset and re-analyzed them using identical FlowDroid configurations. This temporal analysis enables evaluation of mobile logging privacy practice evolution and identification of trends in sensitive data exposure across application lifecycle updates.

In the following chapter, we present the results of these empirical analyses and discuss their implications for Android application privacy practices.

Chapter 4

Case Study Result

In this section, we present the study results of the research questions.

1 RQ1: Which third-party logging libraries are most associated with privacy leaks?

1.1 Motivation

Privacy leakage detection is criticla to ensure user data protection in Android applications, yet current research exhibits significant limitations in scope and coverage. Prior research on privacy leakage detection has primarily focused on small application datasets and concentrated exclusively on Android's official logging utilities [10, 30], while neglecting third-party logging libraries such as Timber, SLF4J, and Firebase in modern Android development. Despite these third-party logging frameworks being extensively used across thousands of applications due to their enhanced functionality and developer-friendly features, they remain largely overlooked in privacy leakage analysis. In this research question, we conduct the first large-scale systematic investigation of privacy leakages through third-party logging libraries. Identifying the praviacy risks shall provide empirical evidence of their impact on mobile application security.

1.2 Approach

We used the validated leaking dataset produced in the prior steps, which contains 1,324 verified leak records across 820 applications. We first determined which applications contained both sensitive sources and third-party logging sinks, forming the analyzable subset of apps. Within this subset, we measured the proportion of applications flagged by FlowDroid as leaking. This app-level prevalence estimates the likelihood of privacy leakage when logging functionality is present, providing a baseline for understanding the scope of the problem.

We then aggregated leak records by the invoked logging API and mapped them to one of the ten libraries in our curated sink list. After filtering out false positives and normalizing method identifiers, we calculated the distribution of leaks across libraries and compared these frequencies to each library's presence in the analyzable dataset. This allows us to see whether privacy risks are concentrated in a small number of frameworks or more evenly spread, and sets the stage for later RQs that examine logging levels, data-flow complexity, and longitudinal changes.

1.3 Results

Overview of FlowDroid Analysis Outcome. Table 4.1 demonstrates the outcomes of FlowDroid analysis. When we applied FlowDroid to each application, we not only saved the taint reports for applications with detected leakages, but also recorded each FlowDroid analysis processing log simultaneously. After analyzing the entire dataset, we categorized the outcome logs into 8 different types:

- **Leak (820 applications):** Applications where FlowDroid successfully identified privacy leakage flows from sensitive sources to logging sinks. These represent the primary focus of our study, as they demonstrate actual privacy vulnerabilities in Android applications.
- **No Leak (843 applications):** Applications that were successfully analyzed by FlowDroid but showed no privacy leakage paths between our defined sources and sinks. These applications either do not use logging libraries or implement proper privacy protection measures.
- **No Entry (14 applications):** Applications that lack a valid entry point for static analysis, typically due to missing or malformed AndroidManifest.xml files or main activity declarations.

- **No Sink Found (36,518 applications):** Applications where FlowDroid could not identify any of our defined logging library methods (sinks) in the application code. This represents the majority of applications that do not utilize the logging frameworks examined in our study.
- **No Source Found (4,355 applications):** Applications where FlowDroid could not detect any sensitive data sources as defined in our source collection, indicating these applications may not access sensitive user information.
- **No Manifest (5,435 applications):** Applications with missing or corrupted AndroidManifest.xml files that prevent proper analysis initialization and component identification.
- **Other Error (574 applications):** Applications that encountered various analysis errors during FlowDroid execution, including memory issues, parsing errors, or other technical failures that prevented successful analysis completion.
- **Overtime (42 applications):** Applications that exceeded the 10-minute timeout threshold during analysis, typically due to complex code structures or excessive computational requirements for taint propagation analysis.

Table 4.1: Overview of FlowDroid Analysis Outcome

Type of Result	**Number of Applications**
Leak	820
No Leak	843
No Entry	14
No Sink Found	36,518
No Source Found	5,435
No Mainfest	5435
Other Error	574
Overtime	42
Total Applications	**48,702**

From Table 4.1, we can observe that leak and no-leak applications represent only a very small percentage of the entire dataset. Leak applications account for 820 (1.7%), while no-leak applications comprise 843 (1.7%). This finding aligns with previous research by [27], which indicated that logging in mobile applications is not as prevalent as in server

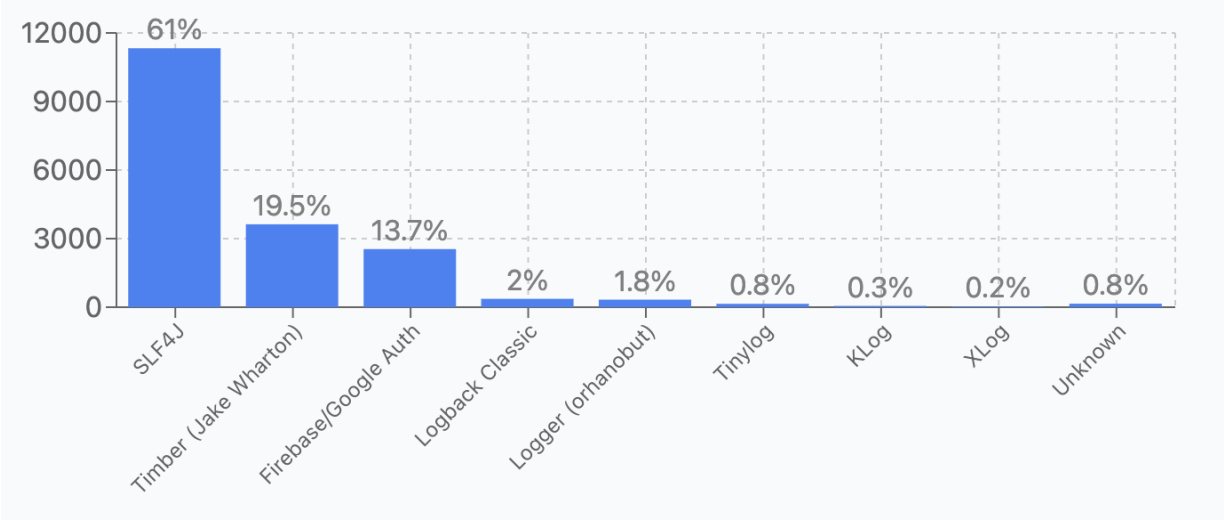


Figure 4.1: Distribution of logging libraries across whole vulnerabilities.

or desktop applications due to several factors including resource constraints and different debugging practices. The majority of applications (36,518 or 74.3%) fall into the "No Sink Found" category, confirming that FlowDroid cannot locate logging libraries in most Android applications. However, among applications where both source and sink methods are present (820 + 843 = 1,663 applications), we observe a concerning pattern: 49.3% of these applications (820 out of 1,663) are flagged as having privacy leaks by FlowDroid. Furthermore, Figure 4.1 demonstrates that mobile logging is predominantly concentrated among three frameworks: SLF4J, Timber, and Firebase. This distribution suggests that platform security measures and developer awareness efforts should prioritize these specific logging libraries to address potential privacy leakage issues more effectively.

Finding 1

Most applications in Google Play do not use third-party logging libraries, confirming that logging is not pervasive in mobile applications. However, when an application does contain logging methods and accesses sensitive data sources, it has a high probability (49.3%) of causing a privacy breach. SLF4J, Timber, and Firebase are the most popular third-party mobile logging libraries.

Data Cleaning and Logging Library Distribution In this section, we eliminated false positives and unrelated records, then analyzed the distribution of logging libraries across all verified leaking records. In Figure 4.2, among the total 18,586 logging records identified

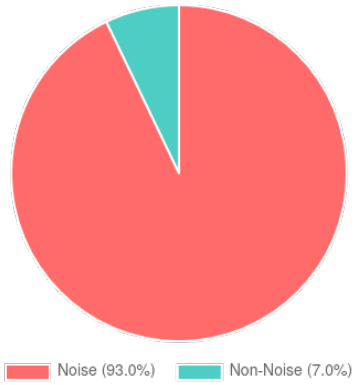


Figure 4.2: Overview of Leaking records.

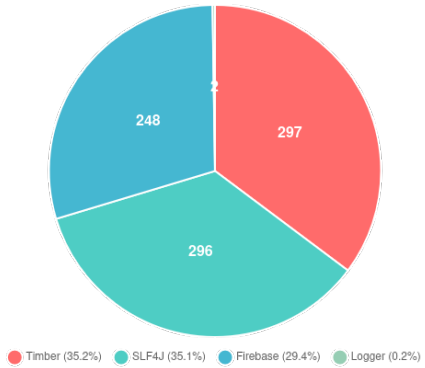


Figure 4.3: Overview of log libraries across Valid leaking records.

by FlowDroid, 17,262 records (93%) were classified as irrelevant records and subsequently removed from our analysis. The remaining 1,324 records (7.0%) represent valid privacy leakage instances that form the basis for our logging library distribution analysis.

Subsequently, we evaluated the logging library distribution across these valid records. As shown in Figure 4.3, only 4 logging libraries were identified across all valid leaking records: Timber with 297 instances (35.2%), SLF4J with 296 instances (35.1%), Firebase with 248 instances (29.4%), and Logger with 2 instances (0.2%). This distribution reveals that Timber and SLF4J are nearly equally prevalent in privacy-leaking applications, together accounting for over 70% of all valid leakage instances.

Finding 2

Timber, SLF4J, and Firebase are the logging libraries most frequently associated with user privacy leaks. While Timber is the most prevalent third-party mobile logging library in privacy-compromised applications, it only marginally exceeds SLF4J usage by 0.1 percentage points, indicating nearly equivalent adoption rates among applications that leak sensitive data.

RQ1 Summary

Our analysis of 48,702 Android applications from Google Play reveals that third-party logging libraries have limited adoption, with only 3.4% of applications containing both logging methods and sensitive data sources. However, among applications that do implement logging functionality, nearly half (49.3%) exhibit privacy leaks, indicating a significant security risk when logging is present. Three logging libraries dominate the privacy leak landscape: Timber (35.2%), SLF4J (35.1%), and Firebase (29.4%), collectively accounting for 99.7% of all verified privacy leakage instances. The near-identical prevalence of Timber and SLF4J suggests equivalent security risks across these popular frameworks. These findings highlight the critical need for enhanced privacy protection measures in widely-adopted logging libraries, as their integration significantly increases the likelihood of sensitive data exposure in mobile applications.

2 RQ2: Which logging levels and data sources most frequently expose sensitive data?

2.1 Motivation

Not all logging statements are equally likely to result in privacy violations. Developers frequently rely on DEBUG or INFO-level logging for development and troubleshooting, but when left enabled in production, these statements can inadvertently leak sensitive user data. Similarly, data coming from user interfaces (e.g., text fields, search boxes) may carry personal input, while system APIs may reveal device identifiers or account information, each posing different privacy risks when logged. By analyzing which log levels and source categories are most associated with leaks, we can better distinguish between accidental debug dumps and systematic data collection, leading to more precise developer guidelines,

safer logging defaults, and improved automated detection rules that target the highest-risk scenarios.

2.2 Approach

To answer this research question, we conducted two complementary analyses: (1) logging level distribution analysis across all valid leaking records for each logging library, and (2) source distribution analysis comparing user input-related sources versus SUSI framework sources among all valid leaking records.

Logging Level Distribution Analysis: We examined the distribution of logging levels (DEBUG, INFO, WARN, ERROR, etc.) according to the official documentation from SLF4J[1] and Timber[2]. For Firebase, which employs a different logging paradigm based on event tracking rather than traditional log levels, we categorized Firebase logging into two distinct levels: Analytics events and Performance Monitoring events. This classification reflects Firebase's unique architecture while enabling comparative analysis with traditional logging frameworks. This analysis reveals which logging levels are most susceptible to privacy violations and whether certain libraries exhibit distinct logging patterns that may contribute to data exposure. By examining level-specific distributions, we can identify whether privacy leaks are concentrated in particular logging categories (e.g., debug information vs. error reporting) and assess the risk profiles associated with different logging practices across third-party frameworks.

Source Type Distribution Analysis: We categorized all identified sources into two primary groups: user input-related sources (derived from GUI components, text fields, and user interaction methods) and SUSI framework sources (Android API methods for accessing sensitive system information). This classification enables us to determine whether privacy leaks primarily originate from direct user interactions or from system-level data access patterns.

2.3 Results

SLF4J Log Levels As illustrated in Figure 4.4, the most frequently occurring log level in SLF4J is INFO with 106 instances (35.8%), followed closely by ERROR with 82 instances (28.0%) and DEBUG with 78 instances (26.4%). These three logging levels collectively

[1] https://www.slf4j.org/
[2] https://github.com/JakeWharton/timber

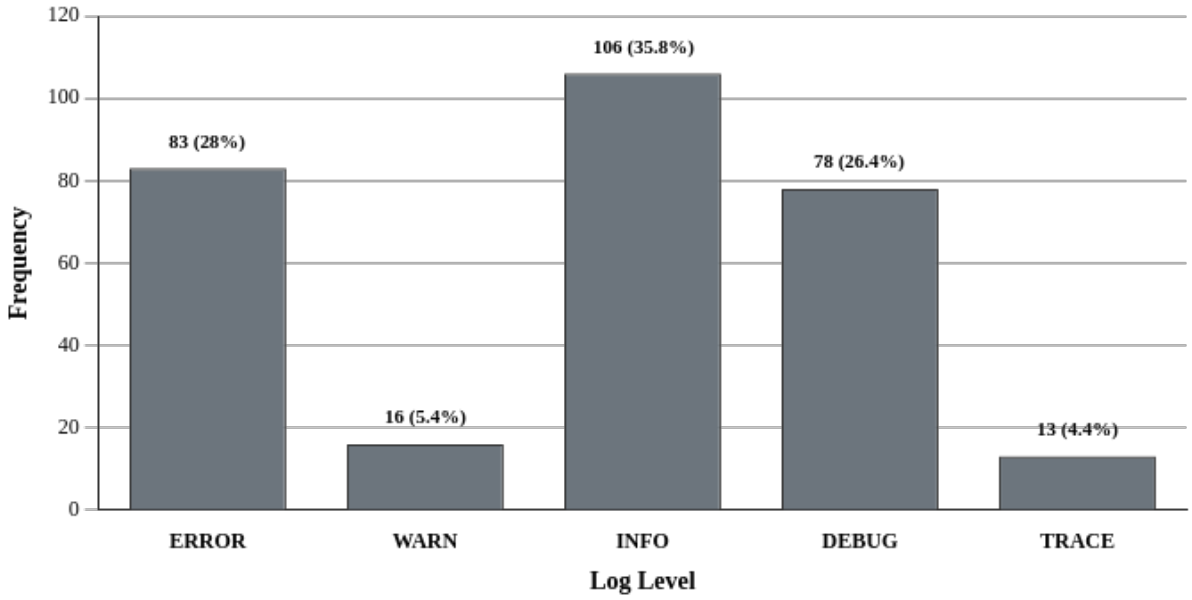


Figure 4.4: SLF4J Log Levels.

account for 90.2% of all SLF4J privacy leaking records, indicating that the majority of privacy violations occur within these specific logging categories. This distribution suggests that developers and researchers should prioritize these three log levels when implementing privacy protection measures and conducting security assessments of SLF4J-based logging implementations.

Timber Log Levels Figure 4.5 demonstrates a markedly different distribution pattern compared to SLF4J. The most frequent logging level is DEBUG with 233 instances (78.5%), followed by ERROR with 48 instances (16.2%). INFO accounts for only 15 instances (5.1%), representing a significantly smaller proportion than observed in SLF4J. TRACE appears once in the dataset, while no instances of FATAL or WARN levels were identified in the analyzed records. This distribution reveals that Timber-based privacy leaks are predominantly concentrated in DEBUG-level logging, suggesting that debugging statements constitute the primary privacy risk vector for this logging framework. The stark contrast with SLF4J's more balanced distribution across INFO, ERROR, and DEBUG levels indicates distinct developer usage patterns between these two logging libraries.

Firebase Log Levels As shown in Figure 4.6, Firebase logging exhibits a highly concentrated distribution pattern. Analytics Events dominate with 243 instances (98.0%), while Performance Monitoring accounts for only 5 instances (2.0%). This overwhelming

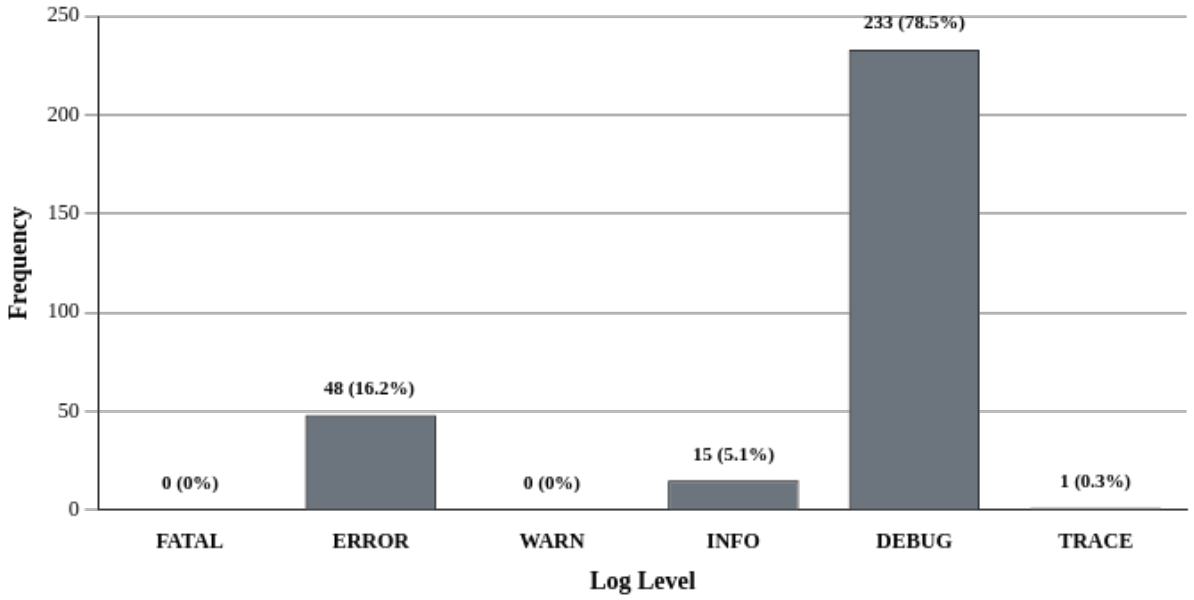


Figure 4.5: TIMBER Log Levels.

prevalence of Analytics Events suggests that privacy leaks in Firebase are primarily associated with user behavior tracking and analytics data collection rather than performance diagnostics. The near-exclusive concentration in Analytics Events indicates that Firebase's privacy risks stem predominantly from its core analytics functionality, where user interactions, preferences, and behavioral data are most likely to be captured and potentially exposed. This finding highlights the importance of implementing stringent privacy controls specifically within Firebase Analytics implementations to mitigate inadvertent data exposure.

Finding 3

Mobile logging libraries exhibit distinct level distribution patterns, with ERROR and DEBUG levels dominating privacy leaks. These two levels account for 54.4% of leaking instances in SLF4J and 94.7% in Timber. For event-based logging frameworks like Firebase, Analytics Events represent the primary privacy risk (98.0%) compared to Performance Monitoring (2.0%). Practitioners should prioritize security measures for ERROR and DEBUG levels in traditional logging libraries, while focusing on Analytics Events rather than Performance Monitoring in event-based logging systems.

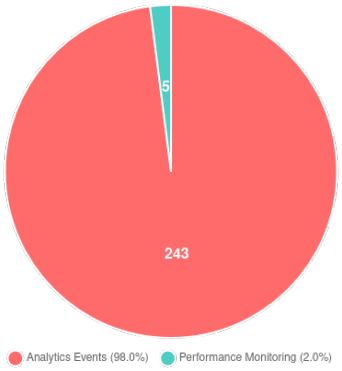


Figure 4.6: Firebase Log Levels.

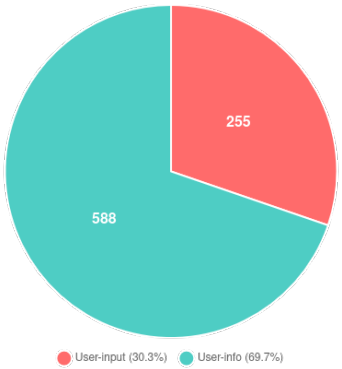


Figure 4.7: Leaking Source Type Distribution

Source Type Distribution According to Figure 4.7, user-info sources account for 69.7% of all leaking records, while user-input sources comprise 30.3% of privacy violations. This distribution reveals that user-input mechanisms represent a significant portion of privacy leakage sources in mobile logging statements, challenging the predominant focus on system-level API sources in previous research [6, 10, 30]. The substantial contribution of user-input sources (30.3%) demonstrates that direct user interactions through GUI components constitute a meaningful privacy risk vector that warrants dedicated attention in mobile application security analysis. While user-info sources remain the majority contributor to privacy leaks, the notable presence of user-input related leakage indicates that developers frequently log sensitive data directly captured from user interface elements, highlighting the need for enhanced awareness regarding input handling privacy practices.

Finding 4

User-input sources represent a significant component of mobile logging privacy leaks, accounting for 30.3% of all leaking records. This substantial proportion demonstrates that direct user interactions through GUI components constitute an important privacy leakage vector that needs to be considered in mobile application security assessments.

RQ2 Summary

Our analysis reveals distinct logging patterns across frameworks: SLF4J shows balanced distribution among INFO (35.8%), ERROR (28.0%), and DEBUG (26.4%) levels, while Timber heavily concentrates in DEBUG (78.5%). Firebase demonstrates overwhelming dominance of Analytics Events (98.0%) over Performance Monitoring (2.0%). Source analysis shows user-info sources dominate privacy leaks (69.7%), but user-input sources represent a substantial 30.3%, highlighting GUI components as significant privacy risk vectors. Practitioners should prioritize ERROR and DEBUG levels in traditional logging libraries, focus on Analytics Events in Firebase, and implement comprehensive privacy controls for user-input mechanisms.

3 RQ3: How complex are the data-flow paths between sensitive data sources and logging statements?

3.1 Motivation

The relationship between where sensitive data originates and where it is logged is not always straightforward. Data often passes through multiple layers of abstraction, transformations, and intermediate variables before being logged. Measuring the complexity of these paths (direct vs. indirect, short vs. long) provides valuable insight into the mechanisms by which privacy leaks occur in practice. If most leaks occur along very short paths, then simple pattern-based or lightweight static checks may suffice to detect them. If, however, they commonly involve longer, indirect flows, more sophisticated static or hybrid analysis tools are required to capture them accurately. Quantifying these path characteristics allows researchers to refine taint-analysis tools, prioritize detection of moderately complex flows where leaks are most common, and guide developers on where to place sanitization or runtime checks.

3.2 Approach

Jimple Code Review and data flow path Analysis: To evaluate the data flow path between logging statements and source methods, we analyzed Jimple code across leaking cases and categorized the data flow relationships into two distinct types: **direct** connections, where logging statements have immediate access to source methods without intermediate processing steps, and **indirect** connections, where data flows through one or more intermediate variables or method calls before reaching the logging statement. Due to the computational complexity of analyzing indirect connections and the challenges in accurately measuring path lengths through multiple intermediate transformations, we focused exclusively on computing the path length for logging statements and sources that have direct connections.

Direct connection type: Based on Listing 4.1, the source from `DBActivationManager.cursorToActivation()`, which extracts an end date string from a database cursor. This data is then passed to `virtualinvoke $r1.<Activation: void setEndDate(...)>($r3)` which is our identify source method before the activation object is returned via `return $r1`. Overall, from source to the intermediate object is 3 statements.

Listing 4.1 Direct type Jimple Code Example of Source Method

```
// com.ascona_locarno.my.MainActivity
private com.ascona_locarno.my.object.Activation cursorToActivation(android.database.Cursor)
    {
        r0 := @this: com.ascona_locarno.my.database.DBActivationManager;

        virtualinvoke $r1.<com.ascona_locarno.my.object.Activation: void
  setEndDate(java.lang.String)>($r3);

        return $r1;
}
```

Listing 4.2 presents the corresponding Jimple code for logging statements within the `syncDataPromo()` method, with a total 11 key steps in the sink method.

Listing 4.2 Direct type Jimple Code Example of log statement

```
// com.ascona_locarno.my.database.DBActivationManager
private void syncDataPromo()
{
    // Step 1: Database retrieval
  $r9 = virtualinvoke $r20.<com.ascona_locarno.my.database.DBActivationManager:
  com.ascona_locarno.my.object.Activation getActivation()>();

    // Step 2: Get end date
  $r4 = virtualinvoke $r9.<com.ascona_locarno.my.object.Activation: java.lang.String getEndDate()>();

    // Step 3: Data parsing
```

```
  $r12 = virtualinvoke $r21.<java.text.SimpleDateFormat: java.util.Date parse(java.lang.String)>($r4);

    // Step 4: Date comparison
  $z0 = virtualinvoke $r11.<java.util.Date: boolean after(java.util.Date)>($r12)

    // Step 5: Conditional check
  if $z0 == 0 goto label31

    // Step 6: Database cleanup
  virtualinvoke $r20.<com.ascona_locarno.my.database.DBActivationManager: void deleteall()>()

    // Step 7: State management
  r0.<com.ascona_locarno.my.MainActivity: boolean isExpired> = 1

    // Step 8: Get email
  $r4 = virtualinvoke $r22.<com.ascona_locarno.my.database.DBUserManager: java.lang.String
   getUserEmail()>()

    // Step 9: Bundle creation
  $r23 = new android.os.Bundle

    // Step 10: Data population
  virtualinvoke $r23.<android.os.Bundle: void putString(java.lang.String,java.lang.String)>("email", $r2)

    // Step 11: Firebase log
  virtualinvoke $r16.<com.google.firebase.analytics.FirebaseAnalytics: void
   logEvent(java.lang.String,android.os.Bundle)>("appDeactivation", $r23)

}
```

From the database retrieval steps to the logging statements, we found a total of 11 statements in the sink method. Additionally, the source method contains 3 statements from data extraction to object return. In this case, we successfully established a connection between the logging statement and its source, which we classify as a direct connection. The combined path contains $11 + 3 = 14$ total statements, meaning the data flow path length is 13.

Indirect connection type: In the Listing 4.3, the source method `$r1 = interfaceinvoke $r2.<android.database.Cursor: java.lang.String getString(int)>($i0)` extracts data from a database cursor and passes it directly through `return $r1;` without intermediate processing steps.

Listing 4.3 Indirect type Jimple Code Example of Source Method

```
// tk.drlue.ical.i.e
 public java.lang.String a(java.lang.String)
    {
        $r1 = interfaceinvoke $r2.<android.database.Cursor: java.lang.String getString(int)>($i0)

        return $r1;
    }
```

In Listing 4.4, the logging method `interfaceinvoke $r8.<org.slf4j.Logger: void info(java.lang.String,java.lang.Object)>("Export schedule started: ", $r1);` retrieves data from `$r1 := @parameter0: tk.drlue.ical.e.n;`. However, we cannot establish a direct connection between `tk.drlue.ical.e.n` and the source methods within the current method scope. In this situation, we are unable to build a direct connection between the logging statement and its source method. We classify these cases as indirect connections.

Listing 4.4 Indirect type Jimple Code Example of Sink Method

```
// tk.drlue.ical.services.RunningMuini
private void b(tk.drlue.ical.e.n, tk.drlue.ical.f.b)
    {
        $r1 := @parameter0: tk.drlue.ical.e.n;
        interfaceinvoke $r8.<org.slf4j.Logger: void info(java.lang.String,java.lang.Object)>("Export
  schedule started: {}", $r1);

    }
```

Automated Analysis with Large Language Models: To scale this analysis process across our entire dataset, we manually applied our established analysis methodology using Claude Sonnet 4 [3]. We first demonstrated our manual analysis approach to the LLM using the examples shown in the listings above. Subsequently, we manually fed each leaking record's corresponding Jimple files to the model, prompting it to analyze each logging statement and source method pair following our established methodology. For each record, we provided the LLM with the specific Jimple code for both the logging statement and its corresponding source method, requesting detailed explanations of the tracking process and step-by-step computation of the data flow path length. This manual feeding approach ensures that each analysis follows our precise methodology while enabling us to process the large volume of leaking records that would be impossible to analyze manually. For each leaking record, the LLM generates specific tracing steps and computation results, maintaining consistency with our established manual analysis framework.

3.3 Results

Connection Type: Based on our manual analysis of logging statement and source Jimple code, we found that across all source types, there are 316 direct connection records compared to 527 indirect connections, indicating that 37.5% of all leaking records exhibit direct connections between sources and sinks. Furthermore, when we examine User_input

[3] https://www.anthropic.com/claude/sonnet

and User_info source types separately, both categories show that direct connections remain less prevalent than indirect connections, with User_input at 31.4% (80 out of 255) and User_info at 40.1% (236 out of 588) direct connections respectively.

This pattern demonstrates that indirect data flow paths dominate across all source categories, suggesting that privacy leaks typically involve intermediate processing steps regardless of the type of sensitive data being accessed. The slightly higher proportion of direct connections in User_info sources (40.1% vs 31.4%) may indicate that user information is more likely to be logged directly, while user input data tends to undergo more processing before reaching logging statements.

Table 4.2: Summary of Log Statements and Source Connection Type

Source Type	**Direct Connection**	**Indirect Connection**
All	316	527
User Input	80	175
User Info	236	352

Finding 5

In mobile logging, most sources reach logging statements through multiple intermediate procedures rather than direct connections. Only 37.5% of privacy leaks involve direct data flows, while 62.5% require intermediate processing steps before sensitive data reaches logging statements.

Data Flow Path Length Distribution Table 4.3 shows the distribution of data flow path lengths from direct connection type leaking records. The majority of data flows between logging statements and sources have moderate complexity, with lengths between 2 and 4 statements accounting for 396 cases (75.1%). The shortest flows (length = 1) and the most complex flows (length ≥ 5) combined represent only 24.9% of all cases, indicating that extremely simple or highly complex data paths are relatively uncommon. When examining different source types separately, the distribution patterns remain consistent. User_input data flows also concentrate in the 2-4 range with 129 cases (73.7%), while User_info sources show a similar pattern with 267 cases (75.9%). This consistency across source types suggests that the moderate complexity pattern (2-4 statements) represents a common characteristic of direct data flow paths in mobile logging privacy leaks, regardless of whether the sensitive data originates from user interactions or system information sources.

Table 4.3: Summary of Data Flow path Length

Source Type	**Length = 1**	**1 < Length < 5**	**Length ≥ 5**
All	51	396	80
User_input	18	129	28
User_info	33	267	52

Finding 6

Most connections between logging statements and their sources exhibit moderate complexity, with 75.1% having data flow path lengths between 2-4 statements. Data flow path length distribution patterns remain consistent across different source types, indicating that source type does not significantly influence the complexity of data propagation paths.

RQ3 Summary

Our analysis of data flow path characteristics reveals that mobile logging privacy leaks predominantly involve indirect connections, with only 37.5% of cases showing direct data flows from sources to logging statements. The majority (62.5%) require intermediate processing steps, indicating that privacy violations typically result from complex data propagation rather than straightforward logging practices. Among the direct connections analyzed, data flow complexity follows a consistent pattern across all source types. Most connections exhibit moderate complexity with 75.1% having path lengths between 2-4 Jimple statements, while extremely simple (length=1) or highly complex (length ≥ 5) flows are uncommon (24.9%). Notably, this distribution pattern remains consistent between User_input (73.7%) and User_info (75.9%) sources, demonstrating that source type does not significantly influence data flow complexity. These findings suggest that privacy leak detection tools should prioritize tracking indirect data flows and focus optimization efforts on moderate-complexity paths, as these represent the most common privacy violation patterns in mobile logging implementations.

4 RQ4: How do privacy leaks in third-party logging libraries evolve across application updates?

4.1 Motivation

Privacy leakage detection through logging libraries has revealed significant vulnerabilities in Android applications, as demonstrated in our previous research questions, yet the temporal evolution of these privacy risks remains unexplored. In practice, mobile applications undergo frequent updates to introduce new features, fix bugs, and enhance security measures. However, whether logging-based privacy issues persist or improve over time as applications evolve remains an open question. In this research question, we conduct a longitudinal comparison of applications across multiple versions to evaluate whether logging-based privacy leaks are decreasing over time, which indicates successful privacy protection strategies, or persisting despite these interventions, which would suggest the need for more aggressive measures such as enhanced developer tooling, stricter enforcement, or improved library design. Understanding these temporal trends is essential for assessing the mobile ecosystem's progress in addressing logging-based privacy vulnerabilities and informing evidence-based strategies for future privacy protection efforts.

4.2 Approach

To conduct the longitudinal analysis, we systematically identified applications that exhibited privacy leaks from our RQ1 findings and collected their updated versions for comparison. We extracted the list of applications with confirmed leaking records from the RQ1 outcomes, then downloaded the corresponding newer versions of these applications from AndroZoo's 2023 dataset.

Application Version Matching: We ensured that each application pair consisted of the same package name to maintain consistency across the temporal comparison. Applications that were discontinued or no longer available in 2023 were excluded from the longitudinal analysis.

Consistent Analysis Framework: We applied the identical FlowDroid configuration used in RQ1 to analyze the newer application versions. This includes the same sink and source collections, timeout settings, and analysis parameters to ensure methodological consistency and enable direct comparison of results between the original and updated versions.

4.3 Results

We successfully downloaded 110 applications in their updated versions from AndroZoo. Table 4.4 presents the longitudinal comparison outcomes. The results demonstrate a generally positive trend in privacy practices: 68.2% of applications show decreased privacy leaks in their 2023 versions, while only 11.8% exhibit increased leaks. The remaining 20.0% maintained the same leak count as their earlier versions. This distribution suggests that the majority of Android applications have implemented improved privacy protection measures over time, potentially reflecting enhanced developer awareness of privacy issues, stricter app store policies, or the adoption of better development practices. However, the persistence of privacy leaks in updated versions indicates that privacy vulnerabilities in mobile logging remain a concern that requires continued attention from both developers and security researchers.

Table 4.4: Overview of Applications with Leaking Records Comparing with 2023 Version

Comparison Category	**Number of Applications**
Same leaking count (old_leaking_num = new_leaking_num)	22
Increased leaks (old_leaking_num < new_leaking_num)	13
Decreased leaks (old_leaking_num > new_leaking_num)	75
Total applications compared	**110**

RQ4 Summary

Our longitudinal investigation reveals that privacy leaks in third-party logging libraries do improve with application updates. This positive discovery suggests enhanced developer awareness and improved privacy practices over time, though persistent leaks in some updated applications indicate continued vigilance is needed.

Chapter 5

Threats to Validity

In this section, we present the threats to the validity of our research.

1 Internal Validity

FlowDroid Analysis Limitations. Our taint analysis relies on FlowDroid, which has inherent limitations that may affect result accuracy. Due to the large-scale nature of our study (48,702 applications), we imposed memory (4GB) and time (20 minutes) limits on FlowDroid execution to ensure feasibility. These constraints may cause analysis to terminate prematurely for complex applications, potentially missing some privacy leakages.

Data Cleaning and Filtering. Our data cleaning process involved removing records containing source methods not present in our predefined collections to eliminate false positives from over-tainting. While this approach improves precision, it may inadvertently remove legitimate leakages if our source collections are incomplete or if method signatures have minor variations across library versions.

LLM Assisted Analysis. We employed Claude Sonnet 4[1] to conduct our analytical method across the large-scale dataset. While we performed manual validation on a subset of cases, this approach may introduce limitations in the analysis process due to relying on a single model for automated categorization.

[1] https://www.anthropic.com/claude/sonnet

2 External Validity

Third-party Library Selection. Our selection of ten third-party logging libraries was based on GitHub star ratings, documentation quality, and manual analysis of Android development resources. This approach may exclude popular logging frameworks that lack public GitHub repositories, are region-specific, or have emerged after our selection period. The generalizability of our findings may be limited to applications using these specific logging libraries.

Application Dataset Scope. Our study focuses on Android applications from Google Play Store between 2016-2021, which may not represent logging practices in applications distributed through other channels, enterprise applications, or regional app stores. To mitigate issues related to Android's ARSC file format compatibility, we selected applications from this specific timeframe. Additionally, the temporal scope may not capture recent trends in logging practices or privacy-aware development.

Source Method Coverage. Most of our source methods rely on SUSI [6], which may result in incomplete coverage of user information sensitivity methods. Some sensitive data sources, particularly those introduced in newer Android versions or specific to certain application domains, may not be captured, potentially affecting the comprehensiveness of detected privacy leakages. To address this limitation, we extended beyond SUSI by including 158 manually collected user input-related methods from GUI components.

3 Construct Validity

Data Flow Complexity Measurement (RQ3). Our categorization of data flows as "direct" or "indirect" based on intermediate program statements provides a simplified view of actual code complexity. Real-world data flow complexity involves additional factors such as control flow dependencies, inter-procedural calls, and semantic transformations that our analysis does not fully capture.

Temporal Consistency (RQ4). We conducted a longitudinal leakage distribution comparison in RQ4 using the same sink and source collections for both historical (2016-2021) and recent (2023) application versions to ensure consistency. However, some third-party libraries may have undergone changes between these periods, potentially resulting in some log statements or source methods not being detected in the 2023 testing. Despite these potential detection variations, the consistent methodology enables valid investigation of developer logging practices and privacy leakage trends over time.

Chapter 6

Related Work

In this section, we introduce the related work.

1 FlowDroid Analytics and Mobile Log Analytics

Numerous studies have leveraged FlowDroid's capabilities for various security analyses. Zhang et al. [28] employed FlowDroid with SUSI as their primary tool for static taint analysis, demonstrating its effectiveness in identifying privacy violations across application datasets. Similarly, Luo et al. [20] used FlowDroid with SUSI to conduct a qualitative study of 1,022 real-world applications, further validating the framework's analytical capabilities. Although FlowDroid has been widely adopted as a taint analysis tool, most research applies FlowDroid with SUSI directly without specialized focus on mobile logging library privacy leakage. More recently, Zhou et al. [30] evaluated 5,000 Android applications and revealed that sensitive data leaking occurs through logging mechanisms, highlighting this as an emerging privacy concern. However, their study focused primarily on Android's native logging system rather than third-party logging frameworks. While Harty et al. [15] evaluated the third-party logging library Firebase, their investigation concentrated solely on logging behavioral patterns without examining whether sensitive data leaks through these logging mechanisms. To date, no comprehensive study has investigated privacy leakage through third-party logging libraries in Android applications. To address this research gap, we extracted 10 popular third-party logging libraries and incorporated user input source methods into SUSI's framework. We then implemented these enhanced configurations with FlowDroid to systematically analyze applications from Google Play, en-

abling comprehensive detection of privacy leaks specifically targeting third-party logging mechanisms.

2 User Input Threats to User Privacy

Previous research has identified user input as a significant threat vector for privacy violations in mobile applications. Choi et al. [11] revealed that user input mechanisms may pose substantial threats to user privacy, particularly through input method vulnerabilities. Zhang et al. [29, 28] also indicated that graphical user interfaces (GUIs) represent a category of sensitive sources that is not adequately covered by existing frameworks such as SUSI. While Zhang et al. [29] proposed an approach to detect sensitive information leakage through GUI components, no research has systematically examined the connection between third-party logging libraries and GUI based data collection. This represents a significant gap in understanding how user input data flows through mobile applications and potentially reaches logging mechanisms. The intersection of user input handling and logging practices presents unique privacy risks, as developers may inadvertently log sensitive user provided data during debugging, error reporting, or analytics collection. User input sources include text fields, search queries, form submissions, and other interactive elements that directly capture sensitive user information. Our work specifically targets identifying and characterizing the relationship between logging statements and user input sources, addressing this previously unexplored connection between GUI based data collection and third-party logging framework privacy leakage.

Chapter 7

Conclusion

We provide a large-scale empirical analysis of privacy risks in Android logging practices, examining 48,702 applications to reveal critical security vulnerabilities in mobile development. While third-party logging library adoption remains limited (3.4% of apps), our findings demonstrate that when logging is implemented, it creates substantial privacy risks with nearly half of logging-enabled applications exhibiting data leaks.

The dominance of three libraries, including Timber, SLF4J, and Firebase, in the privacy leak landscape (99.7% of instances) highlights the urgent need for framework-level security improvements rather than application specific fixes. This concentration of privacy risks in widely-adopted frameworks suggests that targeted improvements to these libraries could have significant impact across the entire Android ecosystem.

Furthermore, our analysis reveals that privacy violations primarily stem from complex, indirect data flows (62.5% of cases) rather than direct logging statements, with most leaks occurring through moderate-complexity paths of 2 to 4 statements. The distinction between logging patterns across frameworks, with SLF4J showing balanced log level usage, Timber concentrating on DEBUG levels, and Firebase dominated by Analytics Events, provides actionable guidance for targeted privacy protection strategies.

Encouragingly, our longitudinal analysis demonstrates that privacy leaks do improve with application updates, indicating growing developer awareness of logging security risks. However, the persistence of vulnerabilities in some updated applications underscores the need for systematic privacy protection measures integrated directly into logging frameworks. These findings indicate that continued improvements are needed to detect user privacy leaks among logging libraries.

Future work should focus on investigating comprehensive data flow path generation between source methods and logging statements, and automatically detecting log leakage involving sensitive information or user input in applications. Additionally, research should explore the development of privacy-preserving logging frameworks that can automatically encrypt user data before transmission.

References

[1] Marco Alecci, Pedro Jesús Ruiz Jiménez, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Androzoo: A retrospective with a glimpse into the future. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 389–393, 2024.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4186-8. doi: 10.1145/2901739.2903508. URL http://doi.acm.org/10.1145/2901739.2903508.

[3] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th international conference on mining software repositories*, pages 468–471, 2016.

[4] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. Uiref: analysis of sensitive user inputs in android applications. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '17, page 23–34, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350846. doi: 10.1145/3098243.3098247. URL https://doi.org/10.1145/3098243.3098247.

[5] Steven Arzt. *Static Data Flow Analysis for Android Applications*. PhD thesis, Technische Universität Darmstadt, Darmstadt, 2017. URL http://tuprints.ulb.tu-darmstadt.de/5937/.

[6] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices*, 49(6):259–269, 2014.

[8] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 356–367, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978333. URL https://doi.org/10.1145/2976749.2978333.

[9] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. An empirical assessment of security risks of global android banking apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1310–1322, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380417. URL https://doi.org/10.1145/3377811.3380417.

[10] Zhiyuan Chen. A comprehensive study of privacy leakage vulnerability in android app logs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 2510–2513, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695609. URL https://doi.org/10.1145/3691620.3695609.

[11] Junsung Cho, Geumhwan Cho, and Hyoungshick Kim. Keyboard or keylogger?: A security analysis of third-party keyboards on android. pages 173–176, 07 2015. doi: 10.1109/PST.2015.7232970.

[12] CIPM Etienne Cussol CIPP/E. Google Play Store Privacy Policy Requirements — termly.io. https://termly.io/resources/articles/google-play-store-privacy-policy-updates/. [Accessed 25-05-2025].

[13] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, page 291–307, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 9783642309205. doi: 10.1007/978-3-642-30921-2_17. URL https://doi.org/10.1007/978-3-642-30921-2_17.

[14] Danny S. Guamán, Jose M. Del Alamo, and Julio C. Caiza. Gdpr compliance assessment for cross-border personal data transfers in android apps. *IEEE Access*, 9: 15961–15982, 2021. doi: 10.1109/ACCESS.2021.3053130.

[15] Julian Harty, Haonan Zhang, Lili Wei, Luca Pascarella, Maurício Aniche, and Weiyi Shang. Logging practices with mobile analytics: An empirical study on firebase. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 56–60, 2021. doi: 10.1109/MobileSoft52590.2021.00013.

[16] J. Mojica Israel, Adams Bram, Nagappan Meiyappan, Dienst Steffen, Berger Thorsten, and E. Hassan Ahmed. A large scale empirical study on software reuse in mobile apps.

[17] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering*, 47(12):2858–2873, 2020.

[18] Li Li. Mining androzoo: A retrospect. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 675–680. IEEE, 2017.

[19] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association. URL https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static.

[20] Linghui Luo, Eric Bodden, and Johannes Späth. A qualitative analysis of android taint-analysis results. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 102–114. IEEE Press, 2020. ISBN 9781728125084. doi: 10.1109/ASE.2019.00020. URL https://doi.org/10.1109/ASE.2019.00020.

[21] Christian Schindler, Müslüm Atas, Thomas Strametz, Johannes Feiner, and Reinhard Hofer. Privacy leak identification in third-party android libraries. In *2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ)*, pages 1–6, 2022. doi: 10.1109/MobiSecServ50855.2022.9727217.

[22] Christian Schindler, Müslüm Atas, Thomas Strametz, Johannes Feiner, and Reinhard Hofer. Privacy leak identification in third-party android libraries. In *2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ)*, pages 1–6, 2022. doi: 10.1109/MobiSecServ50855.2022.9727217.

[23] Weiyi Shang, Meiyappan Nagappan, and Ahmed E Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, 2015.

[24] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 25–36, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884855. URL https://doi.org/10.1145/2884781.2884855.

[25] Peter Story, Sebastian Zimmeck, and Norman Sadeh. Which apps have privacy policies? In Manel Medina, Andreas Mitrakas, Kai Rannenberg, Erich Schweighofer, and Nikolaos Tsouroulas, editors, *Privacy Technologies and Policy*, pages 3–23, Cham, 2018. Springer International Publishing. ISBN 978-3-030-02547-2.

[26] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can we trust the privacy policies of android apps? In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 538–549. IEEE, 2016.

[27] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, 24:3394–3434, 2019.

[28] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, and Jianwei Niu. Condysta: Context-aware dynamic supplement to static taint analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 796–812, 2021. doi: 10.1109/SP40001.2021.00040.

[29] Xueling Zhang, John Heaps, Rocky Slavin, Jianwei Niu, Travis Breaux, and Xiaoyin Wang. Daisy: Dynamic-analysis-induced source discovery for sensitive data. *ACM Trans. Softw. Eng. Methodol.*, 32(4), May 2023. ISSN 1049-331X. doi: 10.1145/3569936. URL https://doi.org/10.1145/3569936.

[30] Rui Zhou, Mohammad Hamdaqa, Haipeng Cai, and Abdelwahab Hamou-Lhadj. Mobilogleak: A preliminary study on data leakage caused by poor logging practices. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 577–581, 2020. doi: 10.1109/SANER48275.2020.9054831.

[31] Yifan Zhou. An automated pipeline for privacy leak analysis of android applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1048–1050, 2021. doi: 10.1109/ASE51524.2021.9678875.