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

Yixi Zhao*, Kundi Yao*, Yiming Tang†, Weiyi Shang*
*University of Waterloo, †Rochester Institute of Technology
y232zhao@uwaterloo.ca, kundi.yao@uwaterloo.ca, yxtvse@rit.edu, wshang@uwaterloo.ca

***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 indicate that while only 3.4% of applications use third-party logging, nearly half (49.3%) of logging-enabled apps exhibit privacy leaks, with three dominant libraries, such as Timber, SLF4J, and Firebase, accounting for 99.7% of violations. Analysis reveals that privacy leaks predominantly stem from indirect data flows (62.5%) with moderate complexity (2-4 statements), originating primarily from user-info sources, although user-input sources represent a substantial risk. Distinct logging patterns emerge across frameworks: SLF4J shows balanced log level distribution, Timber concentrates heavily on DEBUG levels (78.5%), and Firebase is dominated by Analytics Events (98.0%). Longitudinal analysis demonstrates improving privacy practices over time (68.2% of apps reduced leaks), though persistent vulnerabilities underscore the need for systematic protection measures. This study identifies logging-based privacy risks in Android applications and provides actionable insights for developers and library maintainers. It highlights the critical need for practitioners to address both user information and user input as significant privacy threats when using third-party logging frameworks in Android applications.**

***Index Terms*—Mobile Systems, Software Logging, Privacy Leakage, Taint Analysis**

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. Despite the Google Play Store mandating privacy policies and regulatory compliance [10], significant privacy violations persist throughout the ecosystem, with studies finding that 66% of popular applications contain ambiguous, inconsistent, or compliance issues [11].

Among various privacy risk vectors, logging represents a particularly concerning yet understudied mechanism. Logging is a central engineering practice used to diagnose errors, monitor systems, and track processes [14, 12, 20], and is especially valuable in mobile development where many faults surface only on user devices in production environments. While mobile logging serves as an important development tool, it simultaneously poses threats to user privacy, with existing research demonstrating that logs can contain significant privacy vulnerabilities [26, 8].

The privacy risks 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 [13]. 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 show that third-party libraries can contain privacy violations leading to sensitive data leakage [7, 19], 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 [27, 5, 25, 18, 21], with static analysis tools like FlowDroid [4] proving effective for tracking sensitive data flows. However, most existing work either focuses on Android's native logging API or considers logging only as a minor category of data sinks, with a 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 [5] 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, we conduct 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 [4], using an extended source list that includes both system-provided sensitive APIs from SUSI [5] and GUI-based user-

2640-7574/26/$31.00 ©2026 IEEE
DOI 10.1109/SANER67736.2026.00057

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.

The main contributions of this study are as follows:

- 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.
- 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.
- Actionable empirical insights and implications for developers, library maintainers, and tooling designers, including which libraries and logging practices are most associated with leaks, the typical structural patterns of sensitive data flows, and whether issues persist in newer app releases.

Paper organization. This paper is structured as follows: Section 2 introduces the background in Flowdroid, taint analytics, and Android logging libraries. Section 3 describes the subjects we have studied and how we extract the related data. Section 4 explains the motivation, approach, and results of each research question. Section 5 presents threats to validity. Section 6 discusses the related work. Finally, Section 7 concludes this research.

2. Background

In this section, we introduce the Flowdroid and taint analytics, and Android logging libraries.

A. 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 [16]. 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 [4, 6].

Figure 2 illustrates the general taint analysis workflow using FlowDroid according to previous research [4, 25, 24]. 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.

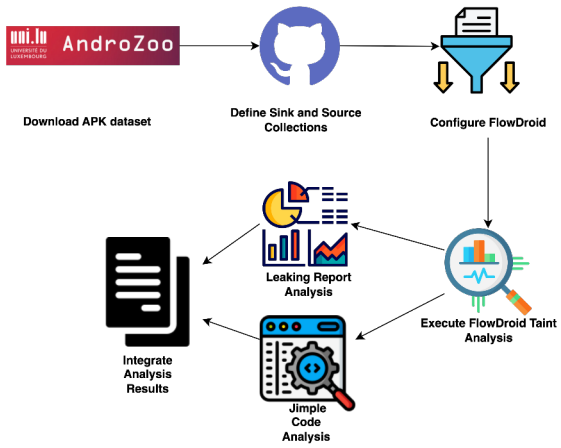


Fig. 1: Overview of Flowdroid workflow

B. Android logging libraries

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

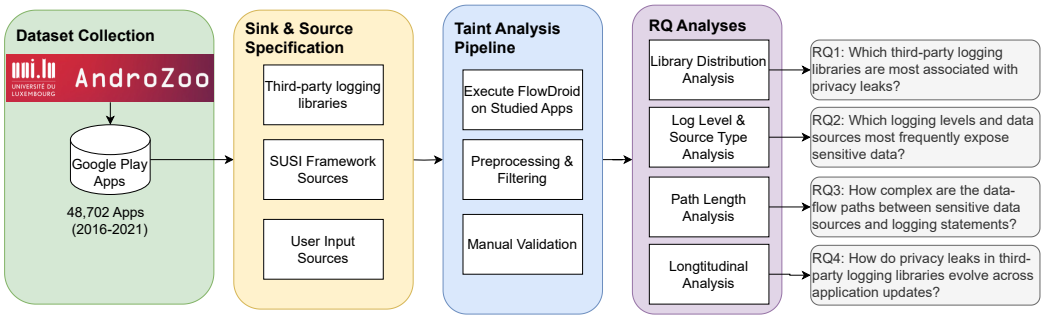


Fig. 2: Overview of our research workflow.

3. Case Study Setup

Our approach comprises four stages, as illustrated in Figure 2: dataset collection, sink and source specification, static taint analysis, and RQ-specific analyses. This pipeline ensures reproducibility and systematically addresses our research questions (RQ1–RQ4) through progressive data refinement and structured analysis at each stage.

A. Dataset collection

We utilize the AndroZoo dataset[2], a comprehensive repository containing millions of Android applications from various sources, including Google Play Store, F-Droid, and alternative markets [2]. This dataset has been extensively used for malware detection, privacy analysis, and large-scale Android research [3, 15, 1]. We focus exclusively on Google Play Store applications for four key reasons: (1) it represents the dominant distribution platform with the vast majority of Android installations, providing the most representative sample; (2) its stricter quality assurance makes privacy violations particularly concerning and broadly impactful; (3) mandatory privacy policies create a controlled environment for examining gaps between stated and actual practices [10]; and (4) over 50% of applications contain privacy policies [22], providing sufficient ground truth.

Through AndroZoo's RESTful API, we downloaded 48,702 Google Play applications spanning 2016-2021, with the majority from 2019-2021. We exclude post-2021 applications due to FlowDroid compatibility issues with newer Android API versions, including evolving source/sink specifications and bytecode format changes [4].

B. 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.

Identify Sink Libraries. Building upon prior research by [26], 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 manual analysis of GitHub [3] repositories and Android Arsenal [4], we identify 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.

Identify Source Libraries. To enable FlowDroid to identify data flows from sensitive inputs to logging sinks, we construct a comprehensive source specification combining both system- and user-level data sources. First, we adopt the SUSI framework [5], which provides a curated set of Android API methods that access sensitive system information. Second, to capture user-provided data, we extend this list with user-input related sources. We systematically collect methods from the official Android API documentation[5] 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 I.

Extract Sink Methods. We systematically extract 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 method counts ranging from 6 methods for Firebase to 91 methods for Tinylog, as detailed in Table I. Each method was categorized according to its logging level (e.g., debug, info, warn, error) and functionality

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

[3]https://github.com/

[4]https://web.archive.org/web/20250206183038/https://android-arsenal.com/

[5]https://developer.android.com/reference

TABLE I: Summary of Sink and Source libraries used in FlowDroid analytics.

Type	Library	Stars	Methods	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

(e.g., standard logging, event logging, HTTP inspection).

Extract Source Methods. We construct our source collection from two primary components: 46 sensitive Android API methods from the SUSI framework [5] 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 II.

Formatting. Following data collection, we manually format all 621 API methods (417 sink methods and 204 source methods)

TABLE II: 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

according to FlowDroid's sink and source pattern specifications [4] to ensure compatibility with the static analysis engine. This formatting process involves converting method signatures to FlowDroid's required syntax, specifying parameter types, and defining appropriate access modifiers for each method entry.

C. Taint analysis pipeline

Next, we conduct taint analysis on the collected applications using FlowDroid [4], and preprocess the analysis results for subsequent empirical evaluation.

Flowroid Execution. To investigate data leakage issues in logging methods, we leverage FlowDroid to perform taint analysis on applications downloaded from AndroZoo [2], using the sink and source collections generated in the previous step. We execute 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 run four parallel threads to analyze applications concurrently. During the analysis process, we record both FlowDroid leakage reports and runtime log files for comprehensive evaluation. For applications with confirmed leaks, we additionally extract 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 implement a multi-stage preprocessing pipeline to clean and filter the results to ensure analytical validity. First, we parse FlowDroid reports into a unified schema, merging per-app results into a single structured dataset. Next, we apply 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 enables 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 perform 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 use LLM to summarize long traces and highlight the final sink calls, with all inclusion decisions verified 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.

D. Empirical Analyses

Based on the preprocessed FlowDroid results, we conduct four targeted analyses:

Library Prevalence Analysis (RQ1). We analyze the distribution of logging libraries across validated leaking records to identify which frameworks pose the greatest privacy risks.
Logging Pattern Analysis (RQ2). We examine log levels (from library documentation) and categorized sources into user information (system data from SUSI) and user input (GUI interactions) to understand how logging severities correlate with sensitive information exposure.
Data Flow Complexity Analysis (RQ3). We analyze Jimple files to categorize flows into direct and indirect connections, then measure path lengths to evaluate data flow complexity patterns between sensitive sources and logging sinks.
Temporal Evolution Analysis (RQ4). We conduct a longitudinal comparison using 2023 versions of applications that exhibited privacy leaks in our 2016-2021 dataset, applying identical FlowDroid configurations to assess whether leaks persist or improve over time.

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

4. Case Study Result

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

Motivation Privacy leakage detection is critical 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 [8, 26], 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 privacy risks shall provide empirical evidence of their impact on mobile application security.
Approach We use the validated leaking dataset produced in the prior steps, which contains 1,324 verified leak records across 820 applications. We first determine which applications contained both sensitive sources and third-party logging sinks, forming the analyzable subset of apps. Within this subset, we measure 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 aggregate 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 calculate the distribution of leaks across libraries and compare 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.

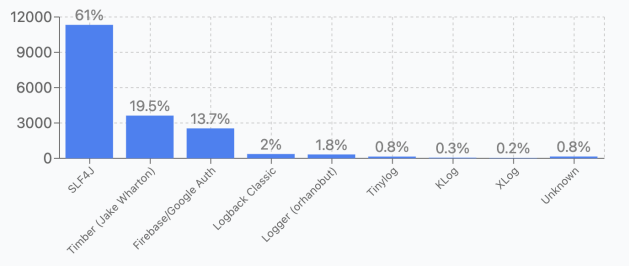


Fig. 3: Distribution of logging libraries across whole vulnerabilities.

Results When we apply FlowDroid to each application, we not only saved the taint reports for applications with detected leakages, but also record each FlowDroid analysis processing log simultaneously. After analyzing the entire dataset, we categorize 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.

We 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

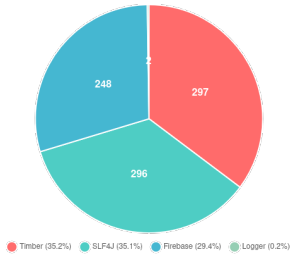


Fig. 4: Overview of log libraries across valid leaking records.

843 (1.7%). This finding aligns with previous research by [23], which indicated that logging in mobile applications is not as prevalent as in server 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 3 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.

We eliminated false positives and unrelated records, then analyzed the distribution of logging libraries across all verified leaking records. Among the total 18,586 logging records identified 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 evaluate the logging library distribution across these valid records. As shown in Figure 4, 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.

RQ1 Summary: Despite limited adoption of third-party logging libraries (only 3.4% of applications), they pose substantial privacy risks when present, with nearly half (49.3%) leaking sensitive data. Three libraries dominate the privacy leak landscape: Timber (35.2%), SLF4J (35.1%), and Firebase (29.4%), collectively accounting for 99.7% of verified leakages. This concentration suggests that targeted improvements to just three frameworks could significantly reduce mobile privacy leaks.

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

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.

Approach To answer this research question, we conduct 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.

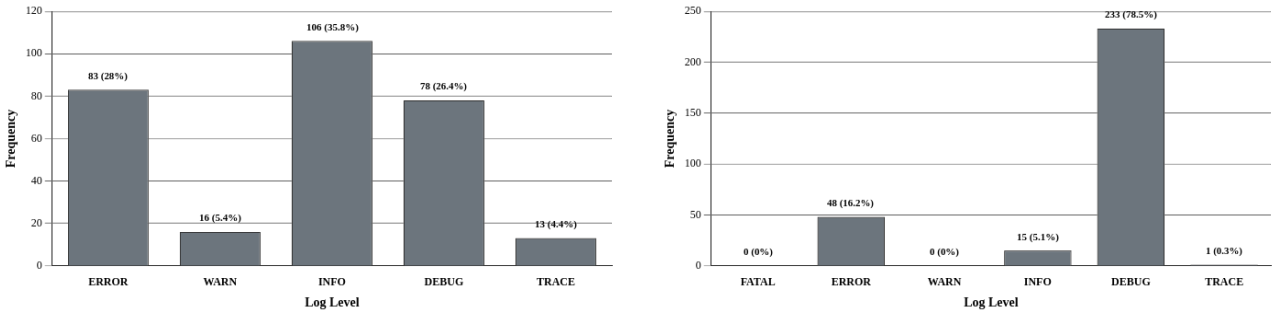
(1) Logging Level Distribution Analysis: We examine the distribution of logging levels (DEBUG, INFO, WARN, ERROR, etc.) according to the official documentation from SLF4J[6] and Timber[7]. For Firebase, which employs a different logging paradigm based on event tracking rather than traditional log levels, we categorize 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.

(2) Source Type Distribution Analysis: We categorize 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.

Results We present the results of our analyses on logging level distributions and source type distributions across all valid leaking records.

Logging Level Distribution. *SLF4J Log Levels*. As illustrated in Figure 5a, the most frequently occurring log level in SLF4J is INFO with 106 instances (35.8%), followed closely by

[6]https://www.slf4j.org/
[7]https://github.com/JakeWharton/timber



(a) SLF4J

(b) Timber

Fig. 5: Log level distributions

ERROR with 82 instances (28.0%) and DEBUG with 78 instances (26.4%). These three logging levels collectively 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 5b 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. 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 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 implementation,s to mitigate inadvertent data exposure.

Source Type Distribution 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 [5, 8, 26]. 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.

> **RQ2 Summary:** Logging frameworks exhibit distinct privacy risk patterns: SLF4J shows balanced distribution across INFO (35.8%), ERROR (28.0%), and DEBUG (26.4%) levels, while Timber concentrates heavily in DEBUG (78.5%). Firebase leaks occur almost exclusively through Analytics Events (98.0%). Notably, user-input sources contribute 30.3% of privacy violations, revealing that GUI components represent a significant yet understudied privacy risk vector compared to traditional system API sources.

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

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.

Approach To analyze the complexity of data-flow paths between sensitive data sources and logging statements, we examined the Jimple intermediate representation code for all verified leaking cases. We categorized the connections between sources and logging sinks into two distinct types:

- **Direct connections**: Cases where we could trace a complete data flow path from the sensitive source method to the logging statement within the same execution context, allowing us to measure the precise path length.
- **Indirect connections**: Cases where sensitive data flows through multiple method calls, class boundaries, or other intermediate program components that prevent straightforward tracing within a single execution context.

For direct connections, we compute the path length by counting the total number of Jimple statements involved in the data flow, from the statement where sensitive data is first accessed in the source method to the statement where it is finally logged in the sink method.

Path Length Calculation Methodology. Our path length calculation for direct connections follows these steps:

1) Count the number of statements in the source method from data extraction to method return
2) Count the number of statements in the sink method from data reception to the logging statement
3) Calculate the connecting points along the data flow path

For example, in Listing 1, the source method `cursorToActivation()` contains 3 statements from data extraction to object return. In the corresponding sink method (Listing 2), there are 11 statements from data retrieval to the final logging statement. Thus, the total path length is $3 + 11 - 1 = 13$.

Listing 1 Direct connection source method example

```
// 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 2 Direct connection sink method example (condensed)

```
// 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()>();

    // Steps 2-10: Data processing and preparation (8
     statements)
    ...

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

In contrast, indirect connections occur when we cannot establish a clear path between source and sink within the analyzed methods. Listing 4 shows an example where the logging statement receives data from a parameter (`r1 := @parameter0: tk.drlue.ical.e.n`), but we cannot directly connect this parameter to the source method in Listing 3 within the current execution context. Due to the computational complexity of analyzing these multi-context data flows, we classified them as indirect connections without calculating their path lengths.

Listing 3 Indirect connection source method example

```
// 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;
    }
```

Listing 4 Indirect connection sink method example

```
// 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);
    }
```

Scaling Analysis with LLM Assistance. To analyze our large dataset systematically, we first establish the methodology through manual analysis of representative examples. We then guide Claude Sonnet 4 [8] to apply this methodology consistently across all identified leaking records. For each record, we provide the LLM with the corresponding Jimple code for both the source and sink methods, along with clear instructions on how to classify connection types and calculate path lengths following our established methodology. This approach enables us to process hundreds of leaking records efficiently while maintaining analytical consistency.

Results *Connection Type:* Based on our manual analysis of logging statements 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 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

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

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 III: Summary of Log Statements, Source Connection Type, and Data Flow Path Length

Type	Category	All	User Input	User Info
Connect Type	Direct Connection	316	80	236
	Indirect Connection	527	175	352
Data Flow Path Length	Length = 1	51	18	33
	$1 <$ Length < 5	396	129	267
	Length ≥ 5	80	28	52

Data Flow Path Length Distribution. Table III 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.

RQ3 Summary: Most privacy leaks (62.5%) occur through indirect data flows requiring multiple intermediate steps rather than direct connections from source to logging statement. Among direct connections, moderate complexity paths (2-4 statements) dominate at 75.1%, with this pattern consistent across both user-input and system API sources. These findings suggest that effective leak detection tools must prioritize tracking complex data propagation paths where sensitive information flows through multiple transformations before being logged.

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

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.

Approach To conduct the longitudinal analysis, we systematically identify applications that exhibit privacy leaks from our RQ1 findings and collect their updated versions for comparison. We extract the list of applications with confirmed leaking records from the RQ1 outcomes, then download the corresponding newer versions of these applications from AndroZoo's 2023 dataset.

Application Version Matching: We ensure that each application pair consists 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 apply 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.

TABLE IV: 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**

Results We successfully downloaded 110 applications in their updated versions from AndroZoo. Table IV 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% maintains 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.

RQ4 Summary: Our longitudinal investigation reveals that privacy leaks in third-party logging libraries do improve with application updates. This positive trend suggests that developer awareness, stricter app store policies, and improved logging practices are gradually addressing privacy issues, though continued vigilance remains necessary as vulnerabilities persist in a significant portion of updated applications.

5. Threats to Validity

Internal Validity. Our taint analysis relies on FlowDroid, which may terminate prematurely for complex applications due to memory (4GB) and time (20 minutes) limits, potentially missing some privacy leakages. Data cleaning involved removing records with unmatched source methods, which may inadvertently exclude legitimate leakages. Additionally, we use Claude Sonnet 4 for automated analysis, which, despite manual validation on subsets, may introduce categorization errors.

External Validity. We analyze ten third-party logging libraries selected based on GitHub metrics and Android development resources, which may exclude region-specific or emerging frameworks. Our dataset, limited to Google Play applications from 2016-2021, may not reflect practices in other distribution channels or recent trends. While we extend SUSI with 158 manually collected user input methods, some sensitive sources specific to newer Android versions or niche domains may remain undetected.

Construct Validity. Our classification of data flows as "direct" or "indirect" simplifies real-world complexity, which involves factors like control flow dependencies and inter-procedural calls. For RQ4, we use consistent sink and source collections across historical (2016-2021) and recent (2023) datasets, but changes in third-party libraries over time may affect detection consistency. Despite these limitations, our methodology enables a valid investigation of logging practices and privacy leakage trends.

6. Related Work

Mobile Logging Analysis and Privacy. Mobile logging has been studied extensively, with Zeng et al. [23] analyzing F-droid applications and concluding that most logging occurs at debug and error levels, often modified during development. Zhou et al. [26] evaluated 5,000 Android applications and found that sensitive data leaks frequently occur through logging mechanisms, though their focus was on Android's native logging system rather than third-party frameworks. Numerous studies have leveraged FlowDroid for security analyses, such as Zhang et al. [24] and Luo et al. [17], who used FlowDroid with SUSI to identify privacy violations in real-world applications. While these studies validate FlowDroid's capabilities, they lack a specialized focus on third-party logging libraries. Harty et al. [12] evaluated Firebase logging but concentrated on behavioral patterns without examining sensitive data leaks. To date, no comprehensive study has systematically analyzed privacy leakage through third-party logging libraries, a gap our work addresses.

User Input Threats to User Privacy. User input has been identified as a significant privacy threat vector. Choi et al. [9] highlighted vulnerabilities in input methods, while Zhang et al. [25, 24] emphasized that graphical user interfaces (GUIs) represent sensitive sources often overlooked by frameworks like SUSI. Although Zhang et al. [25] proposed detecting sensitive information leakage through GUI components, no prior work has systematically examined the connection between third-party logging libraries and GUI-based data collection. This intersection presents unique privacy risks, as developers may inadvertently log sensitive user-provided data during debugging or analytics. Our work specifically targets this unexplored relationship between logging statements and user input sources.

Third-Party Library Privacy Concerns. Third-party libraries, widely adopted in mobile development, introduce inherent privacy risks. Mojica et al. [13] demonstrated that code reuse and framework adoption are common across Google Play applications, while Backes et al. [7] and Schindler et al. [19] showed that third-party libraries can lead to sensitive data leakage through nested dependencies. These findings highlight the complexity of protecting user information in modern mobile ecosystems. Our research extends these works by focusing on third-party logging libraries, a category with unique privacy implications due to their explicit role in data collection and potential transmission to remote servers, providing empirical evidence of their prevalence and privacy impact.

7. Conclusion

We perform 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), they pose substantial privacy risks when present, with nearly half leaking sensitive data. The concentration of 99.7% of privacy leaks within three major libraries (Timber, SLF4J, Firebase) underscores the urgent need for security enhancements at the framework level rather than application-specific fixes. The distinct patterns across logging libraries, such as Timber's focus on DEBUG logs and Firebase's emphasis on Analytics Events, highlight the necessity for tailored privacy-aware logging practices. 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. Finally, our longitudinal analysis shows positive evolution in privacy practices, with 68.2% of applications demonstrating reduced leaks in updated versions. By illuminating the hidden privacy risks in mobile logging practices, our research provides a foundation for developing more secure logging frameworks that balance essential diagnostic capabilities with robust user data protection.

Data availability. The experiment data in this paper are available at https://github.com/senseuwaterloo/Android_Log_Privacy_Leakage.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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].

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.