Catamaran: User Privacy Violation Detection in Mobile Logging

Chenxi Hou
New Jersey Institute of Technology
Newark, New Jersey, USA
ch395@njit.edu

Chun Jie Chong
New Jersey Institute of Technology
Newark, New Jersey, USA
cc255@njit.edu

Zhihao Yao
New Jersey Institute of Technology
Newark, New Jersey, USA
zhihao.yao@njit.edu

Hui Peng
Google Inc.
Mountain View, California, USA
benquike@gmail.com

***Abstract*—Logs are widely used in mobile apps for debugging and diagnosis. Unfortunately, they frequently expose personally identifiable information (PII) due to inattentive logging practices, leading to privacy hazards, which has been categorized as a Common Weakness Enumeration, CWE-532. Existing manual redaction and anonymization techniques are often incomplete and ineffective, as evidenced by the rising number of CWE-532 vulnerabilities. To address this, we introduce Catamaran, a framework to proactively detect PII violations in Android app logs. Catamaran takes a comprehensive approach combining dynamic and static analyses: dynamic analysis captures PII leaks directly at runtime in logcat and other log files, while static analysis expands detection to untriggered code paths by reconstructing contextualized log statements. The static analysis leverages call graph analysis, constant propagation, and Large Language Model (LLM) to intelligently identify potential PII issues. Our extensive evaluation on 3,885 Android apps uncovered runtime PII leakage in 233 apps (approximately 6%), comprising a total of 7,485 violations. The static analysis of 300,073 log statements across 3,994 apps identified 1,788 potential log-based PII leaks in 932 apps and 97 distinct libraries. We have responsibly disclosed our findings, leading to the confirmation of nine vulnerabilities and four patches in the Android Open Source Project (AOSP).**

***Index Terms*—Mobile Privacy, Logging, Program Analysis**

1. Introduction

Logs are widely used for diagnosing and debugging mobile systems by routinely recording various events such as user interactions, network activities, and error states. Logs provide developers with rich information to understand the triggering conditions of specific runtime behaviors, crashes, and performance issues [29], [13], [43]. While logging is an essential practice in mobile app development, it often overlooks user privacy, creating privacy hazards by exposing PII, such as Email addresses, device identifiers [35], and even login credentials [21].

Android provides developers with a logging framework (`logcat`) to write logs to a shared system buffer, which can be accessed by a connected computer or other apps with appropriate permissions. Unfortunately, abusing the shared log buffer is a known attack vector, allowing privileged malicious apps to collect logs system-wide [26], [35]. A recent study [26] found that 81% of pre-installed mobile apps read system-wide logs. As a result, Google recommended that developers should not include any PII in logs [8]. Furthermore, developers can write logs directly to files or upload them to their servers for further analysis with little transparency to users [2]. In response to the privacy concerns, some developers have practiced log redaction and anonymization [3], [5], [10], but these practices are often incomplete, resulting in inadvertent inclusion of PII in logs. As shown in Figure 2, the number of reported CWE-532 vulnerabilities (*"Insertion of Sensitive Information into Log File"*) [1] has increased over the years, indicating a growing concern over the logging of PII.

Prior studies have investigated the issue of PII in logs through static [56] or dynamic analyses [35], [19], [20]. However, they primarily focused on logcat, overlooking app-specific log files. While dynamic approaches like User Interface (UI) fuzzing with logcat monitoring have yielded useful findings [35], [19], [20], their testing coverage is inherently limited. This limitation is significant because developers continuously add, delete, or modify logs throughout an app's development cycle [29], which can increase the likelihood of PII exposure over time.

Furthermore, while prior research on third-party libraries has identified data leakage through channels like networking and profiling [25], [28], [50], [36], these studies have provided limited insight into leakage caused by improper logging practices. PII leakage from library code can propagate to host apps without the developers' knowledge. There is an urgent need for a comprehensive framework that can be integrated into the app development and deployment cycle. Such a framework must be capable of detecting PII leakage in both logcat and other log files, as well as tracing log-emitting statements across both app and library code.

 Digital Object Identifier (DOI): 10.1109/SecDev66745.2025.00013

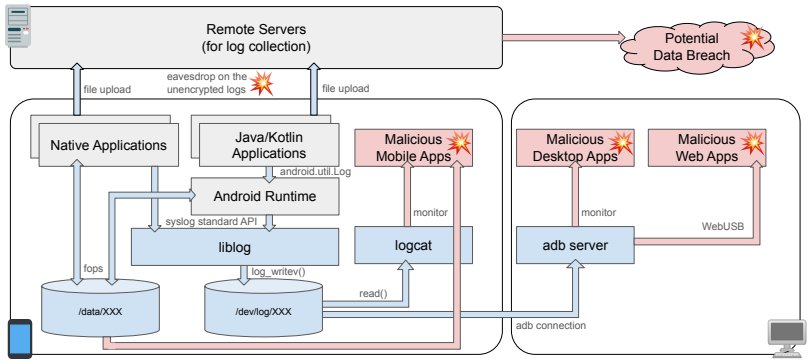

Figure 1: The existing Android logging architecture and its threat model for privacy risks. Blue background indicates the components or paths related to logging, and red background indicates the components or paths that can be exploited.

In this paper, we introduce Catamaran[1] [2], a framework that identifies PII in both system log and log files using a combination of dynamic and static analysis. Our dynamic testing automates installation, execution with random screen events, and detection with logcat monitoring and extended Berkeley Packet Filter (eBPF) [23] tracing. We allow users to specify PII and auto-fill certain device identifiers, and then monitor logs for matches. However, dynamic analysis alone can only reach limited code coverage [51], leading to limited effectiveness in capturing all instances of PII leakage through logs. Thus, Catamaran leverages a novel static approach to complement the dynamic analysis in finding potential issues

Previous static approaches [56] required predefining sources and sinks, but identifying all PII sources is challenging [44], [48], [33], as some sources are from undocumented APIs [45]. To address this challenges, Catamaran utilizes a novel approach in static analysis that does not require predefined sources. Instead, it reconstructs logcat arguments through constant propagation and uses LLM to infer the contextual information that may be printed by logcat at runtime, which we refer to as *logical logcat output*. Furthermore, based on the location of log-emitting statements, Catamaran determines whether PII leakage in a specific log line originates from the app's code or its dependent libraries (see §5.2).

Our main contributions are summarized as follows:

- We present Catamaran, the first framework, to our knowledge, that combines dynamic and static analysis to detect PII leakage in Android app logs, providing leak source attribution. Dynamic analysis identifies runtime PII leakage in logcat and log files, while static analysis proactively detects potential PII exposures. Designed for CI/CD integration, Catamaran enables developers to efficiently detect and fix these vulnerabilities.
- We conduct comprehensive evaluation to showcase the effectiveness of Catamaran. The dynamic part examined 3,885 Android apps and uncovered 233 leaking apps (approximately 6%) with a total of 7,485 PII occurrences in logs.
- The static part of Catamaran analyzed 3,994 apps with 300,073 unique reconstructed logcat outputs. Our study identifies 932 apps and 97 libraries that log PII to logcat. Notably, this includes 15 libraries used widely across more than 10 distinct apps.

We have responsibly disclosed all bugs confirmed during our dynamic analysis to their respective developers, including both third-party apps and Android platform apps and services. As a result of our findings, Google has patched four vulnerabilities in the Android Open Source Project (AOSP) codebase. Additionally, our work has contributed to five additional confirmed Android platform software vulnerabilities, which are awaiting patches. Upon publication, we will open source Catamaran to encourage future research.

2. Background

2.1. Logging on Mobile System

Logging is heavily used for debugging purposes in mobile apps. Android provides a set of standardized logging APIs, namely the logcat framework [11]. As shown in Figure 1, apps can use logcat APIs to write logs directly to the system-wide logging buffer, or use customized logging APIs to write to files.

Logs provide valuable information to developers, but inadvertently logging PII may leak such information to parties with access to logs. Logcat is accessible from all apps with `READ_LOGS` permission prior to Android 4.1, and to pre-installed apps after Android 4.1 [6]. Pre-installed apps provided by OEM vendors are often assumed to be trusted, but this is not always the case. For example, Gamba et al. performed a large-scale study on pre-installed Android apps and found that many of them stealthily collect user data: 81% of the 3,154 studied pre-installed apps read the system-wide logcat buffer [26]. The prevalence of pre-installed apps with excessive log access poses a significant threat to user privacy. Vulnerabilities in these pre-installed platform apps

1. Catamarans are boats with two parallel hulls. Our framework is much like the twin hulls operating in tandem, combining dynamic and static analysis to detect privacy issues in mobile logs.

2. Catamaran is open source: *https://github.com/redoubt-lab/Catamaran*

Vuln ID	Product	Information	Logcat	Storage	Severity
CVE-2019-17395	Third-party app	Username/password	✓		Critical
CVE-2019-17398	Third-party app	Tokens	✓		Critical
CVE-2019-17396	Third-party app	Username/password	✓		Critical
CVE-2019-17394	Third-party app	Username/password	✓		Critical
CVE-2019-17355	Third-party app	Username/password	✓		Critical
CVE-2019-17397	Third-party app	Username/password	✓		Critical
CVE-2023-22362	Third-party app	Credentials		✓	High
CVE-2016-6799	Apache Cordova library	Private information	✓		High
CVE-2017-11134	Third-party app	Login credentials		✓	Medium
CVE-2022-20278	Android Accounts	Account information	✓		Medium
CVE-2018-14995	ZTE Android platform	Phone number and text messages	✓	✓	Medium
CVE-2022-20458	Android CarNotification	Account name	✓		Medium
CVE-2021-0997	Android Location	Access Point names	✓		Medium
CVE-2018-17499	Third-party app	API key and token			Medium
CVE-2018-6599	Orbic Wonder Android platform	Messages/calls	✓	✓	Medium
CVE-2018-15001	Vivo Android platform	Logs of other apps and kernel	✓	✓	Medium
CVE-2021-39715	Android kernel	Kernel memory and addresses			Medium
CVE-2020-0018	Android InputFlinger	User contents	✓		Medium
CVE-2020-0476	Android Notification	User contents	✓		Medium
CVE-2021-0549	Android Bluetooth	Bluetooth MACs	✓		Medium
CVE-2024-21668	React-native-mmkv library	Encryption keys	✓		Medium
CVE-2019-5634	Third-party app	IoT device communication histories		✓	Medium
CVE-2021-39739	Android ArrayMap	SMS content	✓		Low
CVE-2024-40096	Third-party app	PII	✓		Low
CVE-2019-9277	Android ProcFS	App/browser activity	✓		Low
CVE-2021-0991	Android Bluetooth	MAC addresses	✓		Low

TABLE 1: Recent vulnerabilities of containing sensitive information in logs on the Android platform.

may further allow an arbitrary unprivileged app to access logcat contents (e.g., CVE-2018-15001).

Besides, developers can also write logs directly to files. Logs written to app-specific storage are accessible by the app itself and privileged system apps, including the ones installed by OEM vendors. Logs written to shared storage are accessible by all other apps system-wide. Apps may transmit log files to their developers for further analysis. The storage and transmission of log files are often unencrypted [2], which exposes the logs to potential eavesdroppers and hackers.

2.2. Limited Effectiveness of Manual Redaction

To facilitate privacy protection, logging API has been extended to support the anonymization of sensitive information that is explicitly indicated by the developers. For example, Android's Bluetooth API provides a `toString()` method to return the device's MAC address with the first 4 bytes masked [10]. However, the usage of such APIs is not mandatory, and developers may overlook them (as we have observed in our study, see §6.2), leading to the leakage of PII in logs. Besides using redaction APIs, developers can also manually anonymize PII to balance privacy with debugging needs, but manual redaction requires even higher awareness from developers.

Moreover, predefined rules for redaction are not sufficient to cover all sensitive information. For example, password is a highly sensitive information that should never be logged. However, due to the random nature of passwords, it is not feasible to redact all passwords using predefined rules. Incidents of Android devices logging passwords or other login credentials have been repeatedly reported, as shown in the CVEs in Table 1.

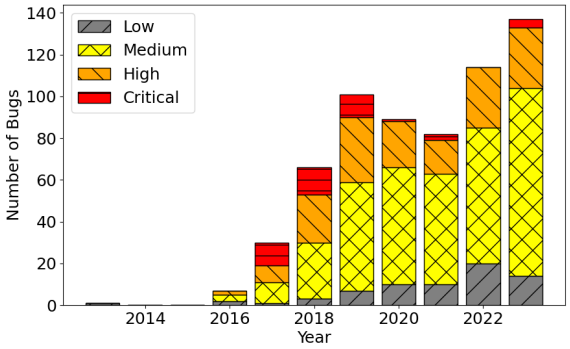

Figure 2: The number of reported `CWE-532: Insertion of Sensitive Information into Log File` in the NVD from 2013 to 2023.

2.3. Developer Guidelines

The outcry for privacy protection in logs is growing. Google has recommended developers to not record PII

and limit log usage in production apps [8]. Besides CWE-532 [1], the CERT Oracle Coding Standard for Java also prohibits the inclusion of sensitive information in logs (FIO13-J [4]) Our study is motivated by the prevalence of missed and incomplete redaction of PII in logs, as evidenced by the increasing number of incidents in recent years. As shown in Figure 2, the number of vulnerabilities reported in the National Vulnerability Database (NVD) has increased significantly from 2013 to 2023. In 2022 and 2023, the reports increased by 39% and 20% respectively. Among them, 32% of the vulnerabilities are rated high or critical severity, and 56% medium severity. The proliferation and severity highlight the urgency of addressing privacy concerns in logs.

3. Motivations and Definitions

3.1. Motivations

Poor logging practices often result in the logging of PII, leading to privacy hazards. Developers introduce new log statements as apps upgrade, and these logs often stay in the codebase even after they are no longer needed, causing a persistent and ongoing privacy hazard in mobile apps [29]. While prior works [56], [35] have studied the PII leakage in logcat, they have limited insight into PII leakage in file-based logs. We conducted a preliminary study by searching for all the CVEs from the NVD [12], with the keyword `Android` and vulnerability category `CWE-532: Insertion of Sensitive Information into Log File`. We manually exclude a vulnerability unrelated to the Android platform, despite the term "Android" appearing in its description. We present the results in Table 1. Among these 26 vulnerabilities, 6 of them (21.4%) dump logs in files. Furthermore, as discussed in §1, third-party libraries are a source of PII leakage that should not be overlooked [25], [28], [50], [36].

3.2. Catamaran Design Goals

Motivated by the observations and our preliminary study, we come up with the following design goals:

G1: Comprehensive Detection. We aim to not only detect all PII that are dynamically logged in logcat and log files, but also identify the potential PII that may be logged from code paths that are not executed during the dynamic analysis. **G2: Fuzzy Match.** In addition to device identifiers and other PII with common patterns, Catamaran also aims to detect unstructured PII, such as approximate GPS coordinates. Fuzzy match is needed to identify GPS coordinates across varying levels of precision. **G3: Identification of leakage sources.** We aim to locate the source of PII leakage in logs, which can help developers and testers understand the root cause of the problem. **G4: Low-Overhead on Mobile Devices.** Although Catamaran is built mainly for software testers, we envision that end users can use it to understand their privacy risks in the daily use of smartphones. Catamaran's dynamic analysis that runs on mobile devices must have low overhead.

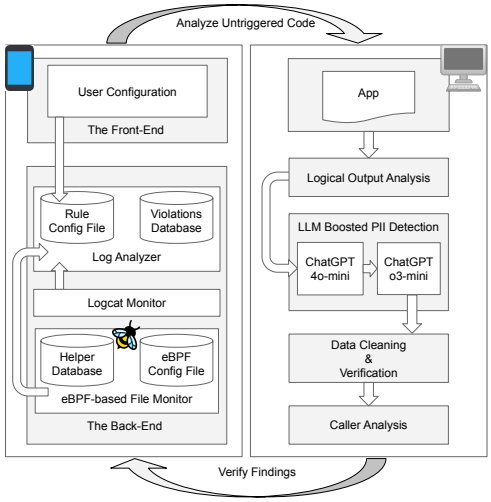

Figure 3: Catamaran features a dynamic analysis component on the left and a static analysis component on the right.

3.3. Threat Model

In our threat model, we assume the intention of the app developer to be benign and the inadvertent logging of PII is a result of overlooking or lack of awareness of such information. We do not consider a malicious developer who attempts to hide or encrypt its logs from users, as such behaviors derail from the purpose of logging, and are out of the scope of this work. However, an adversary may be able to gain privileged access to logs that are not intended for them through means described in §2.1.

4. Catamaran Design

4.1. Overview

Figure 3 shows the architecture of Catamaran, which consists of two parts: an Android app on the left and a static analysis program that runs on a computer on the right. Our Android app is intended to run on an Android device or simulator to monitor logs at runtime. It has a front-end user interface for rule configuration, and a back-end component for monitoring and analysis, which examines all logcat outputs and file writes for potential inclusion of PII. The static analysis program on the right side of the figure complements the dynamic analysis in detecting potential PII in logcat invocations that may not be executed during the dynamic test. This design fulfills **G1** for comprehensive detection.

We give special consideration to certain types of PII in the dynamic test. For GPS coordinates, apps may use varying levels of precision, making it challenging to cover all possible formats. To address this, we adopt a fuzzy match method, which matches approximate GPS coordinates in the logs, allowing for greater flexibility in detecting location data. For PII other than GPS coordinates, we also

encode them using various common hash algorithms for the detection of PII in their hashed forms. The flexibility in PII recognition fulfills our design goal **G2**. Additionally, our log analyzer uses eBPF to ensure low overhead in detecting file changes. This design improves detection efficiency and reduces performance overhead (**G4**).

Catamaran leverages its static analysis flow, shown on the right side of Figure 3, to trace the source of the log statement that emits PII. If a library logs the PII, it poses a more significant risk since the library could be used in multiple apps. To discover such issues, in Catamaran's static analysis, we pinpoint the methods that invoke the logcat API and use their signatures to determine whether they originate from a library or the app itself, thereby achieving **G3**.

4.2. On-Device Dynamic Analysis

Catamaran Android app is responsible for the configuration of PII detection rules and monitoring logs. As discussed in §4.1, our app leverages eBPF for efficient log file monitoring. eBPF is a kernel-mode execution environment that enables secure execution of user-defined programs and provides access to various kernel functionalities, such as network monitoring and file system event tracking [23]. Due to the limited eBPF support in Java/Kotlin, our design separates functionality into two parts: a front-end program written in Java that enables users to configure the system, and a back-end eBPF program written in C, which is responsible for log monitoring and analysis by attaching to write and open syscalls.

4.2.1. Detection Rules Configuration. Although our user interface is designed for developers and software testers, it is intuitive and easy for non-experts to use. Our front-end user interface allows users to provide rules, which are type-value pairs used by Catamaran log analyzer in the back-end to search for PII. We provide a list of potential PII to be completed: device serial number, GPS coordinates, Service Set Identifier (SSID), Basic Service Set Identifier (BSSID), International Mobile Equipment Identity (IMEI), user's phone number, email address, Android Advertising ID (AAID), MAC address, IP address (either in v4 or v6). Catamaran front-end program automatically fills out the fields that can be read from Android system with user's permission, and leave email address, phone number, and IMEI for user input. As discovered in recent research [35], Android apps may log hashed PII rather than plaintext. After users provide their PIIs, Catamaran automatically computes hashes with commonly-used hash algorithms . These rules are stored in a configuration file for later usage in Catamaran back-end program.

4.2.2. Log Monitoring. Catamaran runs a back-end program on the same device under test, which is responsible for monitoring logcat and log files, as well as analyzing them for PII. In our file monitoring approach, we limit the scope to files written by any child processes of the Zygote process within the `/data/` or `/storage/` directories. The Zygote process is a core component of Android that serves as the parent for all app processes. We monitor all app-level activities that may generate log files by targeting the Zygote process and its descendants.

We focus on files with Multipurpose Internet Mail Extensions (MIME) types of `text`, `json`, or `x-ndjson`, and with file suffixes `.txt`, `.log`, or `.text`. If a file's suffix does not match these criteria, we further examine the full file path to determine if it contains the word `log`. Any text file with `log` in its path is considered a log file. If desired, users can customize the scope in an eBPF configuration file.

Catamaran log analyzer in the back-end searches for PII using the rules defined in front-end program. Once a PII is found in a log entry either in logcat or log files, the back-end program stores the related information, such as process name and the violation, to a local database. Due to the design of Android's file system permissions, apps are restricted from reading or writing files outside of their own private directories or designated public directories. This security measure effectively prevents unauthorized access to our violation database and unauthorized access to the PII found by Catamaran in logs. To further enhance security, if needed, we may enforce encryption for the database.

4.3. Static Analysis

We conduct static analysis on app binaries on desktop, focusing on the reconstruction of logcat contents through constant propagation and LLM-based contextual inference. We chose to analyze logcat because it is a standardized logging method, and our dynamic analysis indicates that most PII leakage occurs through logcat. Although PII can also be leaked through file operations, we do not analyze file APIs due to their amount and complexity, which complicates static analysis. If desired, one can still enable such analyses on file operation APIs at the cost of time and memory.

When developers write logging statements, they often embed hard-coded strings. Our key observation is that the propagated string constants provide a good approximation of the logcat output without executing the app. We refer to this approximation as the *logical logcat output*. Specifically, the logical logcat output is a statically reconstructed logcat output in which runtime variables are represented by placeholders, accompanied by contextual information that maps these placeholders to the corresponding PII at runtime. For example, in the logical logcat output shown in Figure 4, the first placeholder can be reasonably assumed to represent a name, and the second represents an email address.

One may also question the false positive of findings in the logical logcat output. We acknowledge that false positives may arise from unreachable or dead code, as well as from logcat statements that appear to contain PII but either do not, or have already been redacted. To answer this, we conduct comprehensive experiments to evaluate the effectiveness of our static analysis in §6.4. Although the static analysis findings may contain false positives, they are still valuable for developers to review and verify.

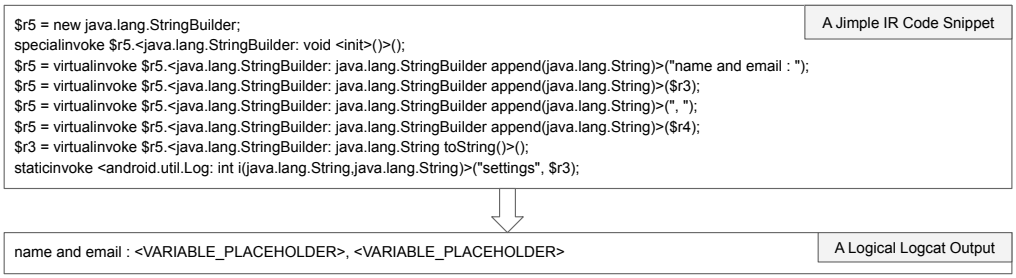

Figure 4: The Jimple IR code snippet shown in the figure is an excerpt from an application.

4.3.1. Reconstructing Logical Logcat Output. Catamaran uses static analysis, including the construction of call graphs and control flow graphs (CFG), for reconstructing and analyzing the *logical logcat output*. A call graph maps the function invocation relationship between methods, and a CFG represents the potential execution paths within a single method. The construction of these graphs is further described in §5.2. We use a depth-first search (DFS) algorithm to traverse all paths in the CFGs of the app under analysis, and output the *paths* that lead to the logcat API calls. Furthermore, to prevent path explosion and memory exhaustion, we cap the number of paths explored per method at 1,000, or limit the exploration time to 3 minutes, whichever comes first. Once either condition is met, analysis for that method stops, and subsequent analysis is based on paths discovered up to that point.

To reconstruct logcat logs, we design a custom approach for iterating through control-flow graphs to propage logging literals through the call paths of interest. Specifically, as shown in Figure 6 in the Appendix, we analyze the parameters of the method being called to reconstruct the logical logcat output. We perform a simulated execution along these identified paths to propagate string constants that are used in the logcat API calls. Our analysis tracks the string-building operations involving string variables, such as concatenation and replacement of string literals along each path. For parameters involving non-string objects or dynamic assignments, their values are represented by a placeholder string.

4.3.2. LLM-Based Contextual Inference. Due to the large volume of logical logcat output collected from apps, manual inspection is not ideal. LLM has shown outstanding performance across a wide range of tasks, including language understanding and reasoning [30], [31], [15]. Given the contextual information in the logical logcat output, LLM is a natural fit for intelligently detecting potential PII. Catamaran uses LLM to examine all logical logcat outputs to determine whether they contain PII. Specifically, it first uses a weaker LLM model to identify potential PII candidates, which are logical logcat outputs that the LLM deems to contain at least one type of PII. It then applies a stronger model to filter out false positives. We prompt the LLM to make a decision by interpreting the semantics of logical logcat outputs and determining whether the variable placeholders correspond to specific types of PII. We present the implementation details in §5.2. This two-step LLM design is for cost reduction, as relying on a stronger model to perform all analyses is ideal but costly. After performing data cleaning and verification (§5.2.1), we conduct a caller analysis to identifies the culprit method (whether from the app or a library).

5. Implementation

In this section, we describe the implementation of Catamaran. We first present the details of our Android app that consists of a front-end and a back-end program for dynamic analysis. We then discuss the static analysis implementation.

5.1. Catamaran Android App

Our Android app consists of a front-end and a back-end program, which both run with root permission. To efficiently analyze logs to detect violations, we combined the eBPF file monitor, logcat monitor, and log analyzer into a single program that serves as the back-end. The front-end program facilitates user configuration of PII values. Upon user input, the program automatically hashes the values (except for GPS coordinates) using six common hashing algorithms for detecting hashed PII in logs.

The back-end program has a eBPF-based file monitor, a logcat monitor, and a log analyzer. The eBPF-based file monitor leverages `libbpf-bootstrap-android` [32] to track system `open` events and system `write` events. Specifically, open events are used to record file paths and inode numbers in a helper database, which can be updated if an old file is deleted and a new file is assigned to that same inode number. This approach avoids costly kernel-space lookups during write events. The helper database is used to associate write events with their corresponding file paths. The eBPF-based file monitor is configurable to specify file types and directories of interest. In addition, we only read new content from a file if its size falls within the configured minimum and maximum thresholds, which are respectively 256 bytes and 2048 bytes. After reading the new content, we update the last read position for that specific file in our helper database and forward the file path along with the content that we've read to Catamaran's log analyzer.

In parallel, our logcat monitor continuously uses the logcat command in root mode to monitor system and app logs, batching 50 lines of logcat prints before sending it to the log analyzer. The log analyzer searches log lines for PII based on rules in the config file, and when a violation is detected, it records the process ID (PID) and uses the *top* command to retrieve the corresponding process name. Violations are stored in a SQLite database [7].

5.2. Static Analysis

Catamaran's static analysis builds on Flowdroid [16] and Soot [52], which are widely used for bytecode analysis in Java and Android apps. Our analysis operates on the Jimple Intermediate Representation (IR), a simplified Java bytecode that facilitates analysis, to reconstruct logcat logs referred to logical logcat outputs. We leverage Soot's default optimizations for dead code elimination and constant propagation [49]. Although more advanced constant analysis methods such as COAL [38], and specialized tools built upon it like IC3 [38] exist, Soot's default implementation offers a practical balance between precision and performance for our objectives.

To extract logical logcat outputs from an APK, FlowDroid [16] generates a call graph for the APK using the CHA algorithm [22], while Soot [52] constructs a control-flow graph for each method within the APK. Subsequently, the algorithm described in §4.3 is then applied to extract all possible logical logcat outputs associated with methods that invoke logcat APIs.

PII detection in the large volume of logical logcat outputs is achieved through a two-stage LLM approach for cost-effectiveness: initially, ChatGPT 4o-mini (2024-07-18 version) [39] performs a preliminary scan to identify potential PII candidates, and subsequently, ChatGPT o3-mini (2025-01-31 version) [40] (7.3x more expensive than 4o-mini) filters out false positives. We provide the same prompt into two models, as shown in the Figure 8 in the Appendix, which includes the types of PII that the LLM is requested to identify, with a placeholder to be replaced by the corresponding logical logcat output.

5.2.1. Data Cleaning and Verification. Human verification is important to ensure result validity and the manual efforts are minimized after using LLM. Therefore we implemented a manual verification process. Specifically, Each logical logcat output flagged by ChatGPT o3-mini as potentially containing PII was manually examined for contextual relevance. We retained only those outputs with clear and sufficient context to confirm the presence of PII, and any ambiguous cases were discarded. This process was conducted by two individuals trained in computer science, and only outputs unanimously identified as containing PII were retained. Although this manual effort may seem extensive, it overlaps with and potentially reduces the workload of regular software tester in locating PII exposure in logs.

Violation Type	Violations In Logcat	Violations In File System
AAID	2,162	24
AAID SHA1	10	0
AAID MD5	11	0
BSSID	97	0
EMAIL	9	7
EMAIL SHA1	6	0
GPS	3,506	94
IPV4	799	75
IPV6	158	0
MAC ADDRESS	53	0
PHONE NUMBER	121	18
DEVICE SERIAL NUMBER	26	0
SSID	305	4
TOTAL	7,263	222

TABLE 2: The number of violations in third-party apps.

5.2.2. Caller Analysis. Based on Android app package naming conventions, a log output is attributed to a library if the invoking method signature includes a package name that does not begin with the app's package name segments. However, discrepancies can occur when developers use package names that differ from the app's package name or when a library uses the same package name. Therefore, to ensure accuracy, we classify a caller as belonging to a library only if its package name appears in at least two apps. This simple approach benefits from our large dataset, and has proven effective in our analysis. Obfuscated package names are excluded from this classification process.

6. Evaluation

We evaluated Catamaran using 4,043 APKs from AppsApk.com [9]. During dynamic analysis, 3,885 apps were successfully tested: 3,864 were third-party apps, and 21 were platform apps or services. Among these, 233 apps (6.00%) were found to log PII: 222 third-party apps and 1 platform app logged PII in logcat, while 14 third-party apps logged PII to the file system. Additionally, we identified violations in 15 types of platform apps and services, and submitted them all to Google, with 5 bug reports currently assigned and awaiting resolution and 4 already patched in the AOSP. Using our static analysis approach, we successfully tested 3,994 apps and identified 932 apps and 97 libraries that would log PII to logcat at runtime. The detailed results of our dynamic analysis, performance evaluations, and static analysis are presented in §6.2, §6.3, and §6.4, respectively.

6.1. Apps Dataset Setup

To construct our dataset, we sequentially crawled APKs from AppsApk.com [9] (a source widely used in prior research [46]), successfully obtaining a total of 6,710 APKs, of which 4,043 APKs could be installed on the Android emulator, forming our final subject dataset. During dynamic and static analysis, 3,885 and 3,994 apps, respectively, were successfully tested. The few analysis failures we encountered were mainly caused by app launch issues during

Platform Applications or Services	IPv4	Mac Address	SSID	Email	BSSID	IPv6	GPS
ActivityTaskManager	•						
Bluetooth		•					
CoreBackPreview			•				
Google Play Service				•			
InputManager			•				
JobScheduler				•			
Location	•	•	•				
Network	•	•	•		•	•	
WI-FI	•	•	•		•		
android.process.acore				•			
com.google.android.apps.fitness				•			
com.google.android.calendar				•			
com.google.android.gm				•			
com.google.android.googlequicksearchbox				•			
com.google.android.inputmethod.latin		•					•

TABLE 3: Violations found in Android platform apps or services.

Library Package Name	PII Data Type	Number of Apps
com.google.android.gms.cast	IP Address	217
com.ironsource	AAID	74
com.millennialmedia	AAID, GPS Coordinates, IP Address, Phone Number	56
com.onesignal	Email	51
com.my.target	AAID (Advertising ID),GPS Coordinates	47

TABLE 4: Top 5 Libraries Ranked by the Number of Apps Logging PII, as Identified via Static Analysis

PII Data Type	Number of Apps
IP Address	373
AAID	315
GPS Coordinates	236
Email	211
Phone Number	129
MAC Address	77
SSID	76
IMEI	42
Device Serial Number	23
BSSID	12

TABLE 5: PII Types Specified in the Prompt and Application Numbers Detected via Static Analysis

dynamic analysis, and issues in generating a call graph or Jimple IR during static analysis.

6.2. Dynamic Analysis

6.2.1. Experiment Setup. We tested all the apps using four emulators simultaneously on a MacBook Pro 2021 laptop with 32GB RAM, 512GB storage, an ARM-based 10-core CPU, and a 14-core GPU. All the Android emulators ran an Android 13 with build number TE1A.220922.012, rooted with rootAVD [37]. The sensitive information searched in the logs was filled out automatically as described in §5.1, except for IMEI, phone number, and email. For these three types of information, we manually fill out a random IMEI, a testing phone number and email address. We wrote a script to automate the testing process. The primary process is as follows: when we test an APK, we first install it and perform dynamic testing using Fastbot2 [34], [42], [18], an ML-based click automation tool that models GUI transitions. We allocate 5 minutes of run time for each app. Then, we fetch the violations from the emulator and uninstall the APK. After testing 50 APKs on an emulator or when an emulator crashes, we re-create a new emulator instance to refresh its states. This process ensures that the testing results are, to the greatest extent possible, not affected by interactions between APKs.

6.2.2. Data Preprocessing. After collecting all results, we removed violations that lacked an identifiable app package name or missed a recognizable platform apps or services name. Besides, we excluded violations involving the IPv4 address `127.0.0.1`, as this address is not considered sensitive information. Furthermore, we cleaned violations in the aforementioned testing automation and root tools, since these tools are designed for testing or research purposes.

While our primary focus is to identify violations in third-party apps, we also detected in platform apps and services. Because platform software often runs continuously, the number of violations tends to increase steadily as we test other apps, making violation counts for platform apps and services an unreliable metric for objective assessment. Therefore, their analyses were separate. Specifically, we manually mapped violations in platform apps and services into their types, such as Wi-Fi, Bluetooth, and location, based on the log content and the process name we obtained to determine which platform apps or services were responsible for the violations.

6.2.3. Violations in Third-Party Application. We count each type of sensitive information identified in a single log line as one violation. If multiple types are found in the same log line, each type is counted as a separate violation. The number of violations can indicate the frequency of a

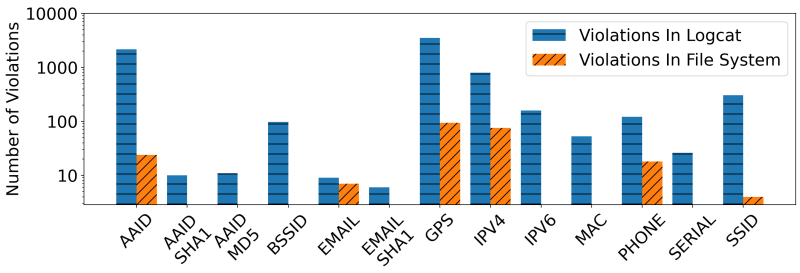

Figure 5: Violations found in third-party apps. MAC means MAC Address; Serial means Device Serial Number.

specific PII logged over a given period. As Figure 5 shows, GPS records appear most frequently in the log, followed by AAID. Both of these data types pose significant privacy risks. We have also found AAID recorded in both MD5 and SHA1 hashed forms and email addresses in SHA1 hashed forms. Although converting the hashes back to plain text is challenging, adversaries may build a dictionary to expedite the hash-cracking process.

As Table 2 shows, the total number of violations in the logcat and file system is 7,263 and 222, respectively. This indicates 97.03% of violations exist in logcat, and the remaining violations are in file-based logs.

6.2.4. Violations in Platform Apps or Services. While our dynamic evaluation mainly focuses on testing third-party apps, we tested 21 platform apps developed by Google, and in addition, our log analyzer monitors the log generated by other system components and preinstalled apps. In the 21 platform apps from AppsApk.com [9] , we found a fitness app that printed the user's email address. For the detailed violations in the platform apps or services found in our dynamic evaluation as shown in Table 3. We reported all of these violations to Google, resulting in 3 bugs being patched in the AOSP codebase. Besides, during the course of this research (but not directly from the evaluation of Catamaran), we discovered and reported 3 additional PII in log in platform services. Consequently, 1 bug has been patched in the AOSP codebase.

6.3. Performance Evaluation

Our front-end program is used to configure the rules for the log analyzer in the back-end, and it is only used at the time of reconfiguration. Catamaran back-end program continuously monitors and analyzes logs in the background. Therefore, we evaluated the performance overhead of the back-end program on a real Android device.

6.3.1. Experiment Setup. We tested the performance overhead on a Google Pixel Tablet with a Google Tensor G2 eight-core processor, 8GB RAM, and 128GB Storage. The system version is a rooted Android 13 with build number TQ3A.230901.001.B1. For the dummy PII, we follow the same way of configuration as mentioned in §6.2.1 We monitor the resource utilization of our back-end program process for 10 minutes, saving the utilization data every second. Within that 10 minutes, we perform normal smartphone usage, such as downloading apps for Google Play, browsing web pages, and using various apps. The user experience does not noticeably degrade during the test.

6.3.2. CPU and Memory Usage. Figure 7 in the Appendix indicates that our program's maximum, minimum, and mean memory usage are 0.20%, 0.10%, and 0.15%, respectively. The maximum, minimum, and mean CPU usage are respectively 172.51%, 2.22%, and 54.15%. Both CPU and memory usage are relatively low compared to their respective upper limits of 800% CPU and 100% memory usage.

6.4. Static Analysis

Catamaran's evaluation were conducted on four computers. Every test on each computer was configured with a maximum JVM heap size of 28GB to keep available RAM unified. We analyzed a total of 300,073 unique logical log outputs. After conducting data cleaning and verification discussed in §5.2.1, we found 932 apps and 97 libraries logging PII in logcat. Table 5 shows the prevalence of PII printed by the apps, such as GPS coordinates, Email, and Phone Number, indicating that logging PII to logcat is a common practice that can cause privacy leakage. Furthermore, Table 4 lists the top 5 libraries that log PII into logcat, ranked by the number of apps that use each library. It indicates that libraries have a broader impact on PII logging in logcat. Since a library can be used by lots of apps, poor logging practices in a library could cause large-scale PII leakage. Out of the total 300,073 unique logical logcat outputs, 5,058 were marked positive by ChatGPT 4o-mini, and 2,572 were marked positive by ChatGPT-o3-mini.

False positive (FP): The analysis process integrates ChatGPT 4o-mini and ChatGPT o3-mini into a unified system. As a result, false positives are determined based on the final step's output from ChatGPT o3-mini. After conducting the data cleaning and verification detailed in §5.2.1, the system produced a total of 1,788 true positive results, of which 784 were false positives.

False negative (FN): We randomly selected 1,000 entries for testing. Using ChatGPT 4o-mini and ChatGPT o3-mini on this sample, we observed 4 and 2 false negatives

determined by the literal meaning of logical logcat outputs, respectively. Additionally, our system detected a total of 1,788 true positives, representing 0.60% of all unique logical logcat outputs. In our sample data, there were 7 true positive entries, accounting for 0.70% of the sample. Therefore, we believe the amount of FN in our analyzed data is small.

7. Related Work

7.1. Static Privacy Analysis

MobiLogLeak [56] statically identified PII-leaking log statements in Android apps using taint flow analysis on logcat. AndroidLeaks [27] is a static analysis framework that mapped Android APIs to required permissions and analyzed ad libraries for PII leaks. POLICHECK [14] statically checked the data flow consistency according to apps' privacy policies. FlowCog [41] leveraged natural language processing (NLP) to determine whether privacy permissions requested by Android apps are justified by data flow semantics. Previous research [44], [48], [33] also points out that obtaining source and sink is challenging in Android. Catamaran shares a similar goal in enhancing mobile privacy through static analysis, but it goes beyond them by providing a comprehensive solution that combines source-free static analysis and dynamic analysis to raise developer awareness of PII logging practices.

7.2. Dynamic Privacy Analysis

Lyons et al. [35] and Chen et al. [19], [20] found widespread PII logging on smartphones. Leveraging random UI testing, their studies monitored runtime logcat outputs at the scale of thousands and hundreds of apps, respectively. Verderame et al. [53] proposed a method combining static and dynamic analysis to identify misalignment between privacy policies and the actual permissions in Android apps. TaintDroid [24], a system-wide dynamic taint tracking and analysis framework for Android, which enables real-time analysis with minimal performance overhead. In contrast, Catamaran's dynamic analysis is based on eBPF and logcat monitoring, and its static analysis complement the dynamic analysis by identifying the root cause of PII logging in libraries, as well as additional potential PII logging locations in the code. Besides, our evaluation leverages ML-based UI testing [34], [42], [18] and has tested nearly 4,000 apps.

7.3. Android Third-Party Library

The security and privacy of dynamic libraries on mobile platform, such as graphics libraries [55], advertisement libraries [36], [17], have been studied. Meng et al. [36] investigated the utilization of user information by major advertising providers for in-app ad personalization on mobile devices. Watanabe et al. [54] examined the origins of mobile app vulnerabilities by identifying buggy libraries. However, these approaches overlook data leakage via libraries and fail to provide a thorough analysis of PII exposure in logs

8. Discussion

While data leakage can also occur through network or non-logging files, Catamaran focuses on detecting PII in log, and future work could extend to detecting PII in other channels. Catamaran does not process binary content and cannot detect encrypted PII. Besides, text file paths containing the substring `log` do not necessarily mean that they are log files, just like file suffixes such as `.txt` and `.text` do not always indicate the actual file type. For dynamic evaluation, we have adopted the state-of-the-art automation tools [34], [42], [18], with the expectation that future research will yield more advanced tools to further maximize the coverage and effectiveness of dynamic analysis.

In our static analysis, we do not distinguish between active code and dead code that may never be executed at runtime, and some log statements in the code may never be triggered during actual execution. Relying solely on the literal interpretation of the logical logcat output for PII detection may overlook PII present in fully dynamically generated logs or PII in logs that lack contextual information. Additionally, it remains uncertain whether the detected PII in the logical logcat output will be redacted at runtime. However, by combining static and dynamic analysis methods, our approach overcomes the limitations of each individual technique while leveraging their respective strengths.

In caller analysis, we do not determine whether a library is used exclusively by a vendor's apps or is available for all developers. Moreover, we do not determine whether the tool library is printing PII, as it may merely act as a wrapper for the actual logcat API. Since Catamaran only associates the package name of callers that directly invoke the logcat API, the PII may be printed directly by the developer via the `util` library. Our evaluation focuses on Java-based Android apps, but Catamaran can use unified native/Java static analysis tools [47] to analyze native code in C/C++.

9. Conclusions

In this work, we present a comprehensive study of mobile logging practice using Catamaran, a framework that combines dynamic and static analysis to detect log information disclosure vulnerabilities in Android apps. Catamaran automatically detects PII in both logcat and log files, and uses static analysis to enlarge coverage and determine the log-emitting locations. Our dynamic study has found 233 disclosures of sensitive information among 3,885 Android apps, including 3,864 third-party applications and 15 Android platform applications or services. Our static analysis of 3,994 Android apps revealed that 932 apps and 97 libraries log PII to logcat.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their valuable feedback. We also thank OpenAI for donating API credits to this work. This work received no external funding.

References

[1] CWE-532: Insertion of Sensitive Information into Log File. https://cwe.mitre.org/data/definitions/532.html, 2009.

[2] Unencrypted Windows crash reports give 'significant advantage' to hackers, spies. https://www.computerworld.com/article/1515493, 2013.

[3] Android Developers - UriCompat - toSafeString. https://developer.android.com/reference/kotlin/androidx/core/net/UriCompat, 2020.

[4] SEI CERT Oracle Coding Standard for Java. https://wiki.sei.cmu.edu/confluence/display/java, 2021.

[5] Android Open Source Project. https://cs.android.com/android/platform/superproject/main/+/main:external/zlib/google/redact.h, 2022.

[6] Android Manifest permission. https://developer.android.com/reference/android/Manifest.permission, 2023.

[7] Sqlite. https://www.sqlite.org/, 2023.

[8] Android developers security checklist. https://developer.android.com/privacy-and-security/security-tips, 2024.

[9] Appsapk.com. https://www.appsapk.com/, 2024.

[10] Bluetoothdevice. https://developer.android.com/reference/android/bluetooth/BluetoothDevice\#toString(), 2024.

[11] Logcat Documentation. https://developer.android.com/reference/android/util/Log, 2024.

[12] National Vulnerability Database. https://www.nist.gov/programs-projects/national-vulnerability-database-nvd, 2024.

[13] Y. Aafer, W. You, Y. Sun, Y. Shi, X. Zhang, and H. Yin. Android smarttvs vulnerability discovery via log-guided fuzzing. In *Proc. USENIX Security Symposium*, 2021.

[14] B. Andow, S. Y. Mahmud, J. Whitaker, W. Enck, B. Reaves, K. Singh, and S. Egelman. Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with policheck. In *Proc. USENIX Security Symposium*, 2020.

[15] Anthropic. Claude 3.5 sonnet model card addendum. https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf, 2024.

[16] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. ACM PLDI*, 2014.

[17] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of android ad library permissions. *arXiv*, 2013.

[18] T. Cai, Z. Zhang, and P. Yang. Fastbot: A multi-agent model-based test generation system. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test (AST)*, 2020.

[19] Z. 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)*, 2024.

[20] Z. Chen, S. S. Deo, P. C. R. Puttaparthi, Y. Tang, X. Zhang, and Shang W. From logging to leakage: A study of privacy leakage in android app logs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024.

[21] National Vulnerability Database. Cve-2019-17395 detail, 2019.

[22] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[23] eBPF.io Community. eBPF Documentation. https://ebpf.io/what-is-ebpf/, 2024.

[24] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. USENIX OSDI*, 2010.

[25] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proc. USENIX Security Symposium*, 2011.

[26] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez. An analysis of pre-installed android software. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[27] C. Gibler, J. Crussell, J. Erickson, and H. 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)*, 2012.

[28] M. C. Grace, W. Zhou, X. Jiang, and A. Sadeghi. Unsafe exposure analysis of mobile inapp advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2012.

[29] S. Gu, G. Rong, H. Zhang, and H. Shen. Logging practices in software engineering: A systematic mapping study. *IEEE Transactions on Software Engineering*, 2023.

[30] D. Hendrycks, C. Burns, S. Basart, A. Critch, J. Li, D. Song, and J. Steinhardt. Aligning ai with shared human values. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[31] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt. Measuring massive multitask language understanding. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[32] JiaHuann. libbpf-bootstrap-android. https://github.com/JiaHuann/libbpf-bootstrap-android, 2023.

[33] M. Kober, J. Samhi, S. Arzt, T. F. Bissyandé, and J. Klein. Sensitive and personal data: What exactly are you talking about? In *Proceedings of the 2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2023.

[34] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang. Fastbot2: Reusable automated model-based gui testing for android enhanced by reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[35] A. Lyons, J. Gamba, A. Shawaga, J. Reardon, J. Tapiador, S. Egelman, and N. Vallina-Rodriguez. Log: It's big, it's heavy, it's filled with personal data! measuring the logging of sensitive information in the android ecosystem. In *Proc. USENIX Security Symposium*, 2023.

[36] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee. The price of free: Privacy leakage in personalized mobile in-app ads. In *Proc. Symp. on Network and Distributed System Security (NDSS)*, 2016.

[37] newbit. rootavd. https://gitlab.com/newbit/rootAVD, 2024.

[38] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[39] OpenAI. Chatgpt-4o-mini. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/, 2024.

[40] OpenAI. Chatgpt-o3-mini. https://openai.com/index/openai-o3-mini/, 2025.

[41] X. Pan, Y. Cao, X. Du, B. He, G. Fang, and Y. Chen. Flowcog: Context-aware semantics extraction and analysis of information flow leaks in android apps. In *Proc. USENIX Security Symposium*, 2018.

[42] C. Peng, Z. Zhang, Z. Lv, and P. Yang. Mubot: Learning to test large-scale commercial android apps like a human. In *Proceedings of the 38th International Conference on Software Maintenance and Evolution (ICSME)*, 2022.

[43] H. Peng, Z. Yao, A. Amiri Sani, D. Tian, and M. Payer. GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation. In *Proc. USENIX Security Symposium*, 2023.

[44] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[45] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *Proc. USENIX Security Symposium*, 2019.

[46] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Service (MobiSys)*, 2016.

[47] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

[48] J. Samhi, M. Kober, A. K. Kabore, S. Arzt, T. F. Bissyandé, and J. Klein. Negative results of fusing code and documentation for learning to accurately identify sensitive source and sink methods: An application to the android framework for data leak detection. In *Proceedings of the 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023.

[49] Soot Program Analysis Framework. Soot command-line options. https://soot-oss.github.io/soot/docs/4.6.0/options/soot_options.html, 2024.

[50] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, 2012.

[51] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach, and J. Klein. Dynamic security analysis on android: A systematic literature review. *IEEE Access*, 2024.

[52] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, 1999.

[53] L. Verderame, D. Caputo, A. Romdhana, and A. Merlo. On the (un)reliability of privacy policies in android apps. In *2020 international joint conference on neural networks (IJCNN)*, 2020.

[54] T. Watanabe, M. Akiyama, F. Kanei, E. Shioji, Y. Takata, B. Sun, Y. Ishi, T. Shibahara, T. Yagi, and T. Mori. Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps. In *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017.

[55] Z. Yao, S. Mirzamohammadi, A. Amiri Sani, and M. Payer. Milkomeda: Safeguarding the Mobile GPU Interface Using WebGL Security Checks. In *Proc. ACM CCS*, 2018.

[56] R. Zhou, M. Hamdaqa, H. Cai, and A. Hamou-Lhadj. Mobilogleak: A preliminary study on data leakage caused by poor logging practices. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020.

Appendix

```
1:  procedure ANALYZE(m, i)
2:      results ← ∅
3:      if Depth > MaxDepth or m ∈ Visited then
4:          return results
5:      end if
6:      Visited.add(m)
7:      for all c ∈ GetCallers(m) do
8:          ser ← ∅
9:          pms ← ∅
10:         results.put(c, ∅)
11:         paths ← GetPaths(c)
12:         seri ← SimExec(paths, pms, i, ser)
13:         if seri ≠ ∅ then
14:             Depth ← Depth+1
15:             rRec ← Analyze(c, seri)
16:         end if
17:         if rRec ≠ ∅ then
18:             for all method ∈ rRec.key() do
19:                 for all pms ∈ rRec.get(method) do
20:                     ser ← ∅
21:                     simExec(paths, pms, i, ser)
22:                     results.get(c).add(ser)
23:                 end for
24:             end for
25:         end if
26:     end for
27:     return results
28: end procedure
```

Figure 6: Pseudocode for analyzing parameter values of a function call.

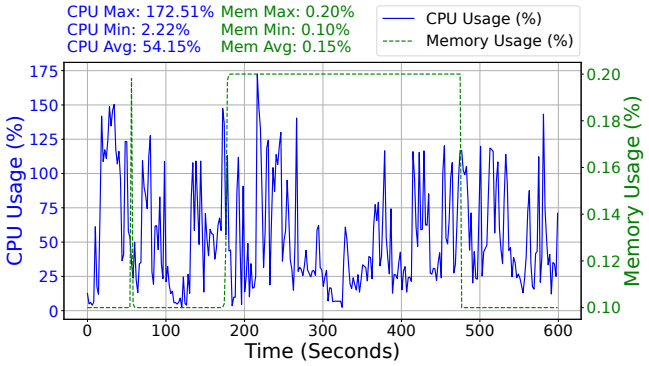

Figure 7: CPU and memory usage in a 10 minute trace.

Analyze the following Android app log to check if the application may print Personally Identifiable Information (PII) at runtime. Specifically, look for <VARIABLE_PLACEHOLDER> entries that could represent any of the following types of PII:

- Device Serial Number
- GPS Coordinates
- SSID
- BSSID
- IMEI
- Phone Number
- Email
- AAID (Advertising ID)
- MAC Address
- IP Address

Analysis Requirements:

1. Only consider PII as being printed if you see <VARIABLE_PLACEHOLDER> in the log.

2. If <VARIABLE_PLACEHOLDER> is present, analyze the surrounding context to determine if it clearly represents one of the PII categories above.

3. If <VARIABLE_PLACEHOLDER> does not appear or there is no clear indication of PII, conclude that the application does not print PII for that category.

4. Avoid over-interpreting or making assumptions. If there isn't enough context to link "<VARIABLE_PLACEHOLDER>" to a specific PII type, mark it as not printed.

Output Format:

Provide your findings in JSON format, with each PII type as a key and two fields:

- printed (true or false)
- explain (a brief explanation detailing why you made this determination)

Log Content: <A logical logcat output>

Figure 8: LLM prompts used for PII Detection.