Log: It's Big, It's Heavy, It's Filled with Personal Data! Measuring the Logging of Sensitive Information in the Android Ecosystem

Allan Lyons
University of Calgary

Julien Gamba
IMDEA Networks Institute / Universidad Carlos III de Madrid

Austin Shawaga
University of Calgary

Joel Reardon
University of Calgary / AppCensus, Inc.

Juan Tapiador
Universidad Carlos III de Madrid

Serge Egelman
ICSI / UC Berkeley / AppCensus, Inc.

Narseo Vallina-Rodríguez
IMDEA Networks Institute / AppCensus, Inc.

Abstract

Android offers a shared system that multiplexes all logged data from all system components, including both the operating system and the console output of apps that run on it. A security mechanism ensures that user-space apps can only read the log entries that they create, though many "privileged" apps are exempt from this restriction. This includes preloaded system apps provided by Google, the phone manufacturer, the cellular carrier, as well as those sharing the same signature. Consequently, Google advises developers to not log sensitive information to the system log.

In this work, we examined the logging of sensitive data in the Android ecosystem. Using a field study, we show that most devices log some amount of user-identifying information. We show that the logging of "activity" names can inadvertently reveal information about users through their app usage. We also tested whether different smartphones log personal identifiers by default, examined preinstalled apps that access the system logs, and analyzed the privacy policies of manufacturers that report collecting system logs.

1 Introduction

Printing diagnostics and informative output messages is fundamental to software development [45]. Meaningful, verbose logging allows one to monitor runtime behaviour and helps to quickly find and diagnose bugs without needing to replicate the issue in a debugger. Continual logging of software in production facilitates monitoring and remediation of runtime crashes. It is therefore typical that software systems log their behaviour in some manner after deployment.

At the same time, logs often contain sensitive and personal information, particularly when logs are unified across a diverse collection of software components. To that point, in 2010 Rosenberg introduced a Linux kernel patch to provide a build option to make access to the kernel ring buffer, i.e., the output of `dmesg`, effectively require root access (`CAP_SYS_ADMIN`); this option was subsequently modified to require `CAP_SYSLOG` and is now widely enabled. In deploying this patch, Rosenberg noted both the futility of attempting to sanitize thousands of `printk` statements along with the resulting loss of functional utility of these sanitized logs [53].

Analogous to Linux, the Android platform also offers a shared system log that collects all logging statements from apps, system services, and drivers [9]. Developers can write to this log using a family of logging functions, differentiated by verbosity, that provide both a log tag and message. A single system log multiplexes the output of all the log statements that occur along with metadata, which includes thread and process identifiers, a log level ("verbosity"), log tag, and timestamp.

On Android, access to the logs has never been part of the published Software Development Kit (SDK) [40], but early versions allowed full access based on requesting a permission (i.e., `READ_LOGS`). Without it, apps can only access their own log messages (but not logs from other apps or operating system components); with the permission, apps have access to *all* log data. Android 4.1 changed the `READ_LOGS` permission to become "privileged," meaning that the permission itself is only available to system apps and apps that come preinstalled (e.g., by manufacturers, carriers, etc.) [14, 40, 41].

Many such privileged apps exist. Android is an open platform that allows any manufacturer to create its own custom version of the Operating System (OS) with preinstalled software that they determine. This means that apps from the phone's manufacturer (OEM) and other key actors in the supply chain, such as Mobile Network Operators (MNOs), OEM partners, and Google, are eligible for privileged access to the system log. Gamba et al. examined preinstalled software in the Android ecosystem [35] and found a vast supply chain with many privileged third parties able to access this data. They noted that many of these apps also included additional SDKs, provided by ad networks, analytics services, or social networks, that inherit the permissions of the embedding app [34] (i.e., these third-party components can read the logs).

The Android Feedback app (`com.google.android.feedback`) is an example of a privileged, preinstalled app. It can launch after another app crashes and gives the user the

```
03-11 12:27:48.110 1801 3775 I LockSettingsService:
  Unlocking user 0 with secret only, length 32
```

Figure 1: A sample log line showing the available fields. The entries include date, time, process ID (1801), thread ID (3775), priority level (I for information), log tag (LockSettingsService), and the log message (the rest of the line) [9].

option to upload their system logs to Google for analysis. This includes all logging, even from apps that are not connected to the crash event. These system logs can also contain arbitrary and varying data: app developers use their own discretion in determining what information to log, though they are given specific guidance by Google not to log private or sensitive information [20]. Some vendors openly disclose their collection of unique identifiers, crash reports, and log data from devices in their privacy policies [38, 43, 54, 68]. Thus, questions remain about the types of sensitive data that are being logged, by whom, and who has access to these logs.

In this work, we present a comprehensive end-to-end study of Android's logging behaviour in practice. The specific contributions of this work are the following:

- We test a variety of stock smartphone models to measure the presence and variation of device and user identifiers that appear in the system log due to the operating system or other preinstalled components.
- We perform a field study to examine the presence of personally identifying information in logs across a variety of users' devices. We show examples of routine logging in particular apps and third-party libraries that can reveal sensitive information.
- We report OEMs who claim to collect and upload personal information and log data in their privacy policies. However, to empirically study how prevalent this practice is across OEMs and device models, we audit a dataset of privileged preinstalled apps gathered by Gamba et al. [35] to quantify how many privileged apps request the READ_LOGS permission and the organizations responsible for them. Then, we use static analysis to study the context around when a system app collects and leaks system logs.

2 Background and Motivation

Android provides app developers a logging system to test and debug the runtime behaviour of their software [20]—a sample log entry is shown in Figure 1. Developers can access the logs in real time over USB or WiFi using Android Studio [15] or via the Android Debug Bridge (ADB) by using the `logcat` command [9]. These tools allow developers to access exception messages, crash logs, and even purpose-specific messages, such as the successful creation of network sockets, user logins, or launched Android Activities. Moreover, the system log also prints routine diagnostics like invoking garbage collection and—when an app throws an exception—the associated stack trace. Developers commonly log messages to verify the correct operation of their software; however, some developers additionally log much more, including sensitive information (contrary to Google's best practices [7]).

Although the device owner can read all log messages using these tools, normal apps only have access to their own logs in order to protect the privacy of data logged by other apps. Only privileged, preinstalled apps can hold the system permission, READ_LOGS [14], that allows the app to read the entire system log, including entries from other apps and the operating system itself. Android's official documentation states that this permission is *"not for use by third-party applications, because Log entries can contain the user's private information"* [14]. The documentation, unfortunately, does not provide a definition of what a third-party application is.

This permission allows device manufacturers and Google to obtain crash reports and other useful information so that they can monitor and debug runtime behaviour. However, incorrect use of the logging system through excessive logging of sensitive information can cause privacy and security harm to users as privileged, preinstalled apps can access any data in the logs and potentially leak this information. For this reason, Google recommends that app developers remove logging statements and the `android:debuggable` attribute from the manifest file prior to releasing an app [17].

Our work is motivated by Reardon's observation that Android devices were logging detailed information from users of the Google-Apple Exposure Notifcation (GAEN) framework [47, 51], including "anonymous" identifiers and diagnoses, which could then appear alongside other identifiable information in the system log. We validated Reardon's findings using a custom COVID Alert app that used the GAEN framework, provided by the Canadian Digital Service. This allowed us to test the logging behaviours surrounding sensitive events—such as reporting positive COVID-19 results and uploading the exposure keys—without triggering alerts on the real system. We found that inferential information was logged to the system log as to whether a user tested positive for COVID-19 and whether they opted to upload their diagnosis. This logging was performed by Google Mobile Services (GMS), indicating that it likely occurred across all Android devices running authorized GAEN apps (i.e., it was not unique to a particular OEM or contact-tracing app). Figure 2 shows an example of this logging.

Moreover, we observed that the Android Open Source Project (AOSP) also logged inferential information regarding the user's COVID-19 status. In particular, AOSP's `ActivityManager`, which is responsible for starting and running apps, would log the name of launched *activities*: user-interface elements, with developer-provided names that may reflect their purpose, within apps. In the con-

```
W ExposureNotification: onClick, accepted: true
[CONTEXT service_id=236]
```

Figure 2: GMS log line indicating a user clicked the "accept" button to share a COVID-19 diagnosis.

```
I ActivityTaskManager: START u0 {act=com.google.
android.gms.nearby.exposurenotification.settings.
SHOW_CONSENT_DIALOG kg=com.google.android.gms
cmp=com.google.android.gms/.nearby.
exposurenotification.settings.SettingsCheckerActivity
(has extras)} from uid 10144
```

Figure 3: A log line printed by AOSP itself indicating it is about to show the SHOW_CONSENT_DIALOG activity to the user.

text of GMS's implementation of GAEN, an activity called SHOW_CONSENT_DIALOG would only be started after a user entered a valid code received from a public health authority after testing positive for COVID-19. Figure 3 shows an example of this logging.

We disclosed our findings to Google, who modified the implementation of GAEN to both stop the relevant logging as well as modified the app so that the logging of activities would no longer reveal a positive diagnosis [26]. Preventing logging of activity names across all Android devices, however, is more challenging due to the diversity of Android variants and versions that are deployed. Thus, the logging of activity names still occurs; later in this work (§5.3) we present case studies on unexpected privacy consequences that this logging can have on users and give recommendations to developers.

3 Related Work

The security considerations of logging systems have long been recognized, such as in RFC 3164, which first documented BSD syslog in 2001 [46]. Logging detailed information during development may be helpful and perhaps even be essential to ensure that the software is behaving correctly, but, in a production environment, this same information can be a liability if exposed to attackers, since observed log messages can provide hints to someone trying to compromise the system [36]. The security risks of logging sensitive information have been documented as a common software weakness in CWE-532 [27], which describes ways in which inappropriate logging can guide potential attackers or reveal user information. This type of software defect has been observed in the wild. For example, as described in CVE-2017-9615, Cognito Software Moneyworks version 8.0.3 and earlier wrote the administrator password to a world-readable file due to verbose logging [24]. This defect allowed attackers to gain administrator access to all data. This defect has also been observed in software that had otherwise been designed for security purposes. CVE-2018-1999036 describes the case where the SSH key passphrase was logged in plain text by an SSH agent plugin [25]. The risk of damage from exposed sensitive log information can be particularly acute when logs can be correlated across a diverse collection of software components, as this correlation may reveal more information than each part separately. Yet, completely anonymous and unlinkable data can be functionally useless.

Zeng et al. previously studied logging practices in mobile apps [69]. They conducted a case study of 1,444 open source Android apps available for download from the F-Droid repository [33] and found that while the majority of logging statements were useful for debugging purposes, excessive logging often still occurred in the release versions of apps, resulting in measurable performance impacts. Additionally, they discovered that developers often chose a logging level inconsistent with the reason for the logging.

Zhou et al. explored the connection between logging and privacy in Android apps [70]. They showed that log statements are common in the release version of many apps and that poor logging practices were connected to the leaking of sensitive data. The leaked data included account names, password information, location information, and device data, such as the MAC address, International Mobile Equipment Identity number (IMEI), and SIM serial number [70].

Despite the potential for introducing privacy and security harm to end users, the research community has not yet performed a systematic analysis of the connection between an inappropriate use of the Android log system by Android app developers—who log Personally Identifiable Information (PII) to system logs despite Google's recommendations—and the wide range of privileged system apps that can read and collect these logs through the READ_LOGS permission.

4 Methodology

We present our research methods in four parts. First (§4.1), we enumerated the types of identifiers that we search for along with the various encodings that we deem equivalent. Second (§4.2), we dynamically executed thousands of apps in an automated testbed to capture the log files generated by those apps, as well as the log files from different stock devices not running apps to capture baseline logging activity from OS components, drivers, etc. Third (§4.3), we attributed different log lines to the responsible entity, such as SDKs, apps, device drivers, etc., by statically analyzing the apps and the AOSP source code. Fourth (§4.4), we crowdsourced system logs from different devices using a privacy-preserving method in order to measure the prevalence of PII in the system logs of a typical user. Fifth (§4.5), we conducted case studies of inappropriate logging by manually examining app behaviour.

4.1 Personal and Device Identifiers

We searched for the following identifiers: (*i*) *direct identifiers*, which are e-mail addresses, phone numbers, and user names; (*ii*) *indirect identifiers*, which are Android IDs, device MAC addresses, IMEI, and serial numbers; (*iii*) *user location*, which are fine- and coarse-grain GPS coordinates—we consider GPS coordinates to be present if both the latitude and longitude are juxtaposed; we consider fine location to have three decimals of accuracy (≈100 m) and coarse location to have two decimals (≈1 km). We also use identifiers relevant for "assisted-GPS," which are the MAC addresses of nearby Bluetooth (BT) devices and the SSIDs and MAC addresses of both connected and nearby WiFi routers. We note that BT MAC addresses can be used to infer users' social graphs, whereas static BT devices can be used for inferring users' geolocation. Note that Android requires that apps hold a location permission to access the MACs and SSIDs of nearby WiFi devices [16] since scan results reveal user location [1, 55].

We also distinguish between a *real* and a *randomized* MAC address. The real MAC address is the one that is provided by the device manufacturer; the former three octets are a prefix that indicate the manufacturer and the latter three indicate the device itself. A randomized MAC address is instead a random MAC-address-sized value that can be used to hide the real MAC address when doing public connections, e.g., associating with an access point. Android uses a new randomized MAC address when first associating over WiFi, but it is only randomly generated when first connecting to a particular SSID. That is, once the MAC address is randomly selected, it continues to be used for connecting to the same access point going forward and so continues to serve both as an identifier and as an indicator of location.

We search for identifiers in log files in a number of ways by removing and altering relevant punctuation and applying standard encodings appropriate to its type. For example, MAC addresses are sought with and without colon separators and with the colons replaced with hyphens. User names and SSIDs are sought normally and with HTTP-safe encoding; strings and hexadecimal sequences are examined without regard for case. We also search for base64 encodings of the identifiers, as well as MD5, SHA1, and SHA256 hashes. While hashes are normally one-way functions, the direct identifiers and forms of location all have small enough domains to make a brute-force preimage search feasible; from a privacy standpoint, we consider them to be the same as sending unhashed variants.

4.2 Dynamic Analysis

We ran a variety of stock unrooted smartphones and captured their system logs. Next, we automated the execution of a variety of apps and collected their system logs. The purpose for both these analyses was to understand what sort of PII may be present in system logs.

Stock Device Testing We gathered logs from real running devices using multiple means. First, we bought a variety of stock devices from different manufacturers, booted them, and performed initial setup, including connecting to WiFi, turning on location and Bluetooth, and signing into a Google account (required to use the Play Store). We then used the system settings to enable developer options and therein enabled USB debugging. We plugged the device into a computer via USB, authorized the computer to access the device, and retrieved the logs with `adb logcat`.

We tested for the possibility that logging may only be triggered on devices with developer options or USB debugging enabled: were that the case, any resultant logging would affect far fewer end users. To test this hypothesis, we found that Samsung devices offered a secret dialer code that would write the current system log to the SD card without needing the device to be in developer mode. We used this code to save the logs to a file which we subsequently emailed to ourselves. This allowed us to reject the hypothesis that developer mode changed logging behaviour. Further conversations with Google regarding our logging concerns gave us no reason to believe that any relevant logging changes occur as a result of developer options or USB debugging being enabled.

App Testing We ran 5,000 randomly-selected apps on an automated testbed and extracted the system logs after each test. The testbed installed each app from the Play Store and used the Android exerciser monkey—a UI fuzzer—to interact with the app for six minutes. After execution of each app, we collected the resulting system logs and searched them for the identifiers listed in §4.1. This process was intended to be a screening process to identify apps at scale that logged certain identifiers without any additional user configuration or interaction; we recognized that this process would yield a lower bound. For example, if an app altered its behaviour based on user configuration or specific user data, this behaviour would not be captured. We report the PII types and log tags found.

From this dataset, we examined in more detail any PII types that appeared in more than 20 of these files with the same log tag. To control noise, no other user-installed apps were run simultaneously on the testbed; however, AOSP components also wrote to the logs. We attributed log entries associated with AOSP components by finding the corresponding tag and message within the AOSP source code. Because our dynamic testing was performed on rooted and instrumented devices, for selected examples we replicated the relevant logging on a stock, unrooted, Pixel 3a running Android 12 to ensure that a normal user would encounter this logging by running one of the test apps. (We report on these results in §5.)

4.3 Attributing Log Tags

We looked at five sources of log entries: (*i*) core system services and development libraries included with AOSP; (*ii*) device drivers necessary to run AOSP on specific hardware;

(*iii*) development libraries and services provided by Google, including GMS; (*iv*) other third-party SDKs and frameworks used by app developers; and (*v*) the apps themselves.

Each log entry has a corresponding log tag that indicates the source of the log entry. In order to attribute log entries to specific apps or to the system, we examined apps and AOSP for tags that we observed occurring in the system log files.

We analyzed the AOSP source code to identify log tags used by the open source components of the operating system and libraries. In most cases, the tags used for logging were defined as constants in either the class or a particular source code file and were identifiable by name (e.g., static final strings named `TAG` or `LOG_TAG`). In other cases, they were defined as constants with different names, but were still identifiable because they were used as a parameter for a call to a standard logging function. Some tags, however, were constructed dynamically to attribute the entry to the caller of the function. For some processes associated with AOSP, the log tag was not explicitly defined and defaulted to the process name.

Analyzing a compiled app is more challenging. Apps may have their own wrapper around Android's logging API and developers may use code obfuscation techniques. Our general approach to extracting log tags from an app was to decode the Android Package File (APK) using `apktool` [67]. We then searched for logging-relevant code in the smali files and tracked where in the code these calls were made.

4.4 Measuring PII in the Wild

We gathered information from real phones using crowdsourcing methods to determine if PII is typically found in device log files. Collecting results from a variety of devices is necessary since different models may have different software loaded by the manufacturer or carriers. In addition, due to regional customization of devices, not all models are the same in all markets around the world and crowdsourcing allows us to collect data on a diversity of models that would not be available to us otherwise. Finally, the choices of apps installed by users will also impact the PII logged.

Using the WebUSB facility built into recent versions of Google Chrome [39], we developed a website and accompanying app that analyzed the phones of volunteers for the presence of PII in system logs. Figure 4 provides an overview. Our IRB agreed with our assessment that this research did not meet the definition of "human subjects research,"[1] and therefore declined to review it. That is, while humans were involved, we were not collecting data *about* human behaviour, nor was human behaviour a focus of our study [52]; we were using humans to crowdsource data from a diverse set of devices and apps. (Even though the data could reveal details about individuals, we did not use it for that purpose, and therefore the study does not meet the legal definition of human subjects research.) Nonetheless, we still applied the same

[1] 45 C.F.R. §46.102(e)(1).

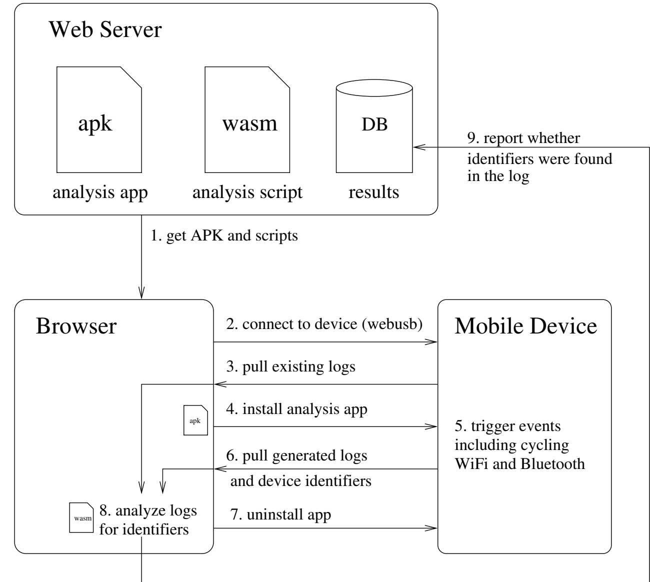

Figure 4: Diagram of our crowdsourced experiment. Volunteers ran the log analysis code on their own devices to detect any PII that might be in the log files. Only a summary of the results were uploaded to our server.

protections that are used in human subjects research: we directed prospective participants to a website that included an overview of the procedure, a consent form, and a privacy policy that included examples of the data to be collected. We thus received informed consent from all participants.

We took great care to not collect any identifiable information from the phones or information that could be used to identify a specific volunteer. This is why we spent many months constructing an online environment using WebAssembly and WebUSB so that raw logs would not leave participants' browsers (i.e., it would have saved a lot of time and given us better data if we had simply uploaded the raw logs).[2] Log files on participants' USB-connected phones were processed by their web browsers, which then transmitted reports to us listing the *types* of data found, rather than the logs themselves.

The *Android Debug Bridge for Web Browsers* library [23] enabled the scripts on our website to use the ADB [12] protocol to access the phone, retrieve identifiers, manage the installation and removal of our analysis app, and access the logs. Our scripts also used ADB commands to toggle the Bluetooth and WiFi radios to exercise the logging that occurs during regular operations, such as establishing network connections. We separately analyzed the logs for PII before and after this intentional exercise of code paths.

[2] While we took steps to not associate the collected metadata with other identifiers, it is certainly possible that unique combinations of app names in conjunction with data from other sources could still identify participants, which is why we are not releasing our data. We believe that all of the claims that we make in this paper can be independently verified without it.

We wrote an Android app that was used by our website scripts to gather identifiers available to user-installed apps, in addition to the ones available with ADB. Depending on the version of Android, some identifiers can neither be accessed through ADB nor through our app. In particular, the IMEI is unavailable in Android 10 and up, the phone number is unavailable for version 11 and up, and only the randomized WiFi MAC is available in versions 11 and up, while the true MAC is completely unavailable [13]. While we could have asked our volunteers to provide such identifiers by navigating layers of settings manually, we opted to minimize participation friction and excluded those identifiers in our analysis.

Logs were processed to determine if any of the gathered identifiers were stored in any log messages. In order to preserve volunteers' privacy, the phone logs were never uploaded to our website. Rather, logs were analyzed by JavaScript and WebAssembly programs run within each user's browser. If PII was detected in a log line, then the programs reported the log tag associated it, the name of the process matching the Process Identifier (PID) stored in that line, and the PII type. Importantly, the specific PII never left participants' browsers and we disabled our logging of the PII to the JavaScript console before deploying our production build.

Upon completion, this data along with generic data, such as model, manufacturer, etc., about the phone were submitted to our server. We then instructed participants to disable developer mode and USB debugging.

We filtered the crowdsourced data to exclude incomplete reports, as well as reports from devices with build variants [8] or signatures [19] that indicated development devices. We excluded development devices because they may have additional, non-standard logging enabled. Beyond these filtering steps, we neither attempted to identify nor excluded counterfeit devices, nor ones that might be rooted. The goal of this part of the study was to identify whether PII is typically found in log files of devices that are being used by users in the field, not necessarily to tie a specific device to a specific manufacturer. Since we did not collect a complete list of the apps that were installed on the devices, we cannot be completely sure what entity was responsible for any given log entry.

4.5 Manual App Inspection

We selectively analyzed apps by executing them manually and then inspected the logs for any potential PII that could be attributed to the app. This approach allowed us to execute apps in a more realistic manner than afforded by the Android exercise monkey, at a cost of not being fully automated and consequently less scalable. However, it did allow us to inspect the logs more creatively and to search for types of PII that might be missed by our automated testbed.

We examined the logging behaviour of 230 Android applications taken from the Google Play Store's top app charts. Given that Android logs the names of "activities" as an app executes, we also examined the list of activities declared in the application manifest to assess whether private information about the user, the device, or their actions may be revealed simply through normal interaction with the application.

5 Results: PII in the Logs

In this section, we present our analysis of the kinds of PII present in Android system logs by both system and user-installed apps using the approaches described in §§4.2 to 4.4. First, we examined a variety of stock, unrooted mobile phones from different manufacturers to dynamically see what PII was logged by default by system components; we then looked more deeply into AOSP to find the logging that was enabled by default. Second, we performed a crowdsourcing experiment to determine what PII was logged in the wild by both user-installed and preloaded apps. Finally, we present specific results from a set of case studies into particular apps and SDKs using manual analysis of the apps while collecting the logs. Our results show that a large variety of PII is typically logged and logs are capable of identifying the user of the device and providing location information.

5.1 Analysis of Default Logging

Between March and June of 2021, we purchased new Android smartphones from Amazon and carrier stores, corresponding to a variety of brands and models (our goal was to optimize for a diversity of devices). We examined each phone in a controlled environment to observe what information they wrote to the system logs; Table 1 gives the results of this experiment. Motivated by the findings of GAEN-related data in the logs, we searched for either emitted or received payloads of Bluetooth Low Energy (BLE) messages in addition to the PII types described in §4.1.

Overall, we found that email addresses were consistently logged across devices (e.g., due to Google Play Services), as well as information about the currently-connected WiFi hotspots. This WiFi information is considered location data [57, 61–63], and it has been shown that location traces are highly unique between individuals [28]. Databases exist to precisely map the locations of WiFi hotspots for enabling network-based geolocation [66]; consequently, Android requires a location permission simply to access information about nearby WiFi routers. Precise GPS coordinates were logged by a majority of devices that we examined. Two devices, the Blu Studio Mini and Nokia 3.4, logged the MAC addresses of nearby BLE devices; the latter also logged the full Bluetooth beacon payload of broadcast messages sent by nearby BLE devices. We shared several of these findings with a subset of the manufacturers in 2021.

We prioritized purchasing inexpensive devices, so that we could acquire a broader sample. However, as a result, many of these devices arrived with older versions of Android (e.g., the

Phone			Identifier										Proximate Data				
Manufacturer	Model	Android Version	GPS	SSID	BSSID	BT MAC	WiFi MAC	IMEI	Serial	AAID	Phone #	Email Address	Nearby SSIDs	Nearby BSSIDs	Nearby BT MACs	BT Payloads	READ_LOGS
Blackberry	Priv	6	✓	✓	✓	✓	✓	✓	✓		✓	✓					1
Blu	Studio Mini	9		✓	✓	✓	✓					✓	✓		✓		5
Cubot	Note 7	10		✓	✓			✓			✓	✓	✓	✓			4
Google	Pixel 3a	9		✓	✓		✓	✓			✓	✓					6
		12		✓	✓						✓	✓	✓	✓			6
Huawei	Nova 5T	9		✓	✓	✓	✓					✓	✓	✓			58
LG	K51	10	✓	✓	✓	✓			✓			✓					4
		12		✓	✓	✓	✓		✓			✓					58
Motorola	G Play	10	✓	✓	✓	✓	✓	✓	✓	✓		✓					34
Motorola	moto g7 plus	10	✓	✓	✓		✓		✓			✓					35
Motorola	One 5G Ace	10	✓	✓	✓	✓	✓	✓	✓		✓	✓					34
		11	✓	✓	✓	✓			✓		N/A*	✓	✓	✓	✓		29
Nokia	3.4	10	✓	✓	✓		✓	✓				✓	✓	✓	✓	✓	5
		12	✓	✓	✓	✓	✓		✓			✓	✓	✓	✓	✓	22
Samsung	Galaxy A12	10		✓	✓	✓			✓			✓	✓	✓			14
Samsung	Galaxy A21S	10		✓	✓							✓					83
		12		✓	✓							✓					95
Sony	Xperia F3113	7	✓	✓	✓		✓					✓					59
uleFone	Note 11P	11		✓	✓	✓						✓					34
ZTE	Blade A5 2020	9		✓	✓		✓		✓			✓	✓	✓			4

Table 1: The various devices we tested from March-June of 2021, including the versions of Android preinstalled and whether they were observed logging various identifiers and proximate data. The last column indicates the number of preinstalled apps that had been granted the READ_LOGS permission. Five phones supported major OS upgrades when we retested them in February 2023, the results of which are depicted on subsequent rows; three devices could not be located for retesting (Huawei Nova 9, Samsung A12, and ZTE Blade A5 2020). (*)This phone was carrier locked and a compatible SIM card could not be found for retesting.

Blackberry Priv with Android 6). By policy, Android OEMs must provide at least one major OS update, which tend to be released annually [22]. Thus, in addition to examining a baseline by examining the sensitive information logged by the phones with the shipped software, we decided in February of 2023 to examine which, if any, of these phones could be updated to new major versions of Android and whether those updates changed the amount of sensitive information logged (especially since we had alerted a few of these manufacturers to these issues almost two years prior).

While two of the phones in Table 1 could not be located for retesting (Samsung A12 and ZTE Blade A5 2020), of the remaining 13, only 5 had major OS upgrades available. Worse, the logging of sensitive information did not appear to decrease in subsequent Android versions:

- Google Pixel 3a: while we did not observe the IMEI or WiFi MAC continue to be logged after upgrading from Android 9 to 12, we observed the additional logging of nearby WiFi networks (SSIDs and BSSIDs) in 12.
- LG K51: GPS coordinates were no longer observed in the logs when upgrading from Android 10 to 12, however, we observed the WiFi MAC address, which was not observed previously.
- Motorola One 5G Ace: we did not observe the IMEI or WiFi MAC continue to be logged when upgrading from Android 10 to 11, however, the upgrade resulted in observations of nearby SSIDs, BSSIDs, and Bluetooth MAC addresses, which were not previously observed.

- Nokia 3.4: we did not observe the IMEI continue to be logged when upgrading from Android 10 to 12, however, the upgrade resulted in observations of the serial number and Bluetooth MAC address in the logs.
- Samsung Galaxy A21S: we observed no changes in the sensitive information logged when upgrading from Android 10 to 12.

Given that a cursory examination of Android devices demonstrated that sensitive information is prevalent in system logs, we decided to look more closely at AOSP logging in general to identify precisely where some of this logging occurred. The open-source nature of AOSP allows us to confirm our findings by attributing the observed logging to the specific line in the source code.

We found many different components that periodically log PII. Two recurring patterns are (*i*) the presence of a static boolean variable called `DBG` that would prevent logging were it `false`, but is instead set to `true`, and (*ii*) classes whose `toString` method includes PII in their output—Java implicitly invokes this function when a class instance is concatenated with a string, e.g., when including a logging statement. We describe and discuss relevant cases next:

`wpa_supplicant` This is used to connect to WiFi networks. In the periodic group rekeying operation, it logs the MAC address of the WiFi router. It also logs the device's own MAC address during initialization of the driver. During initial connection to a WiFi network, it logs the router's SSID when trying to associate and logs the router's MAC multiple times after associating, thus linking the MAC and SSID. The `wpa_supplicant` driver also logs both the router MAC address and the router SSID in the same line for *all nearby* routers during an initial scan. This logging is done by default but can be suppressed by setting the compile-time flag `CONFIG_NO_STDOUT_DEBUG`, which disables the debug messages or `CONFIG_NO_WPA_MSG`, which additionally disables the informational messages. It should be noted that one phone in our collection, a Xiaomi Redmi Note 9 running Android 11, did not log `wpa_supplicant` messages, which suggests that its production code was built with these options set.

`DHCP client` This is used to lease IP addresses from a DHCP server. The implementation has a boolean `DBG` variable that is set by default to `true`, and which guards logging in the class. When logging happens, the received packets are logged using their `toString` method whose implementation logs the randomized MAC address of the mobile device.

`WifiService` This is used to manage the device's WiFi connections, and has multiple log lines that include PII. One is an information message about default gateways that includes the router's MAC address. Another is a log line written when MAC address randomization occurs, which includes both the old and the new MAC address along with the connected router's SSID. Given that the purpose of MAC address randomization is to have a device's MAC addresses be unlinkable, the fact that both the old and new values are logged together undermines that objective.

`BluetoothManagerService` This is used to manage the device's BT connections. It logs the device's *real* BT MAC address in a few places, as well as the device's "name." This name is freely changeable by the device owner, but in many instances has a default value with a structure like "John Smith's phone." A lower-layer component in the hardware project also logs the real BT MAC address in its `get_local_address` function call. Due to nature of assigning MAC addresses during manufacturing, the device's real Bluetooth address and its real WiFi address are often similar; all the phones that we checked differed only in the two lowest-order bits of the last octet. The implementation uses a boolean `DBG` field set to `true`, but were it `false` the logging would stop.

`WifiScoreReport` This manages network performance measurements. If it is not able to start scoring, which happened periodically in our tests and on our stock device, it logs warning messages containing the connected router's SSID and MAC address along with the randomized MAC address of the device.

`GnssNetworkConnectivityHandler` This helps implement "assisted GPS" which users router MAC addresses and SSIDs to help with geolocation [64]. It has a method called `updateNetworkState` that is periodically invoked. When it does so, it logs the SSID of the connected router.

`ConnectivityService` This manages network connectivity on the device. It has a `handleNetworkUnvalidated` method that is periodically invoked, at which point it logs a `NetworkAgentInfo` object. This includes the connected router's SSID and MAC address along with the MAC address of the device. This logging is controlled with a class static `DBG` boolean configured to true. It also has a `makeDefault` method that logs both the router's SSID and its MAC address, also controlled by the `DBG` flag.

`KeyguardUpdateMonitor` This is a component in Android's telephony stack. It has a method that logs the `toString` representation of telephony objects called *SubscriptionInfo*. These include the user's real phone number along with other telephony-related values, such as the mobile network code and mobile country code.

Logging in GMS Finally, we noticed that in many apps the user's email address associated with the phone was logged with a number of different log tags that were not in AOSP. These were `SignInPerformer-X` for some number X, `Backup`, and `CheckAccountFragment`. Using the fact that the system log included the process IDs that caused the logging, we were able to confirm that the relevant processes belong to components of GMS.

Region	Count	Percent
Europe	712	59%
North America	272	22%
Africa	145	12%
South America	39	3%
Oceania	28	2%
Asia	18	1%

Table 2: Participant distribution by continent.

5.2 PII in the Wild Experiment

Inspecting devices for logged PII at scale is challenging. Purchasing every handset model from every manufacturer is costly even without considering further regional variations of devices like customizations set by MNOs and the specific apps that users install. Thus, we turned to crowdsourcing to collect data from a variety of devices around the world to measure whether PII is broadly found on phones.

We primarily recruited participants through the crowdsourcing sites Prolific and Amazon Mechanical Turk; we recruited additional volunteers through our personal networks. Our data collection period ran from March 2, 2022 through March 21, 2022. In total, 1,400 participants submitted reports for 1,405 unique devices comprising 571 model variants from 46 manufacturers running 17 different releases of Android.

We identified unique devices by taking the Android ID, or serial number if the Android ID was unavailable, and used it as the key in an HMAC function to generate the tag for a constant string. Note that this method is how Android IDs are generated as of Android 8, and so using this method ensures that we use the same privacy-preserving approach for devices running Android 7 or lower, or which do not have an Android ID available. We identify model variants from the `ro.product.manufacturer` property reported by the OS. Note that this is a self-reported and not-attested value; it can be bogus, e.g., for a counterfeit device [5, 35, 48, 60].

Since manufacturers have regional customizations for their phones, we also estimated the geographical dispersion of our participants by geolocating the countries corresponding to the IP addresses in our web server request logs to ensure that crowdsourced submissions were not all from a particular region. The results displayed in Table 2 show that just under 60% of our participants were from Europe, with the next most represented regions being North America and Africa.

After filtering out incomplete reports and devices that appeared to be used for development, our crowdsourced data represents 529 model variants from 45 manufacturers. Table 3 shows the number of models and the number of devices running each OS version for the most common manufacturers. The majority of the devices in our dataset were running Android 11 or newer with the most represented manufacturers being Samsung, Xiaomi, and Huawei. The top manufacturers in our sample correlates with statistics available online [59].

Manufacturer	**Model**	**Android Version**					
Name	**Count**	**≤ 7**	**8**	**9**	**10**	**11**	**12**
Samsung	171	17	31	37	63	162	140
Xiaomi	62	2	2	14	63	134	26
Huawei	68	8	17	22	81		
Motorola	42	5	10	14	19	36	1
OnePlus	30			3	12	65	3
Google	14					4	61
Sony	20	5	4	6	3	3	2
Nokia	16			5	4	10	
Realme	11				1	15	1
LG Electronics	13	4	4	1	6	1	
Oppo	12			1	1	12	1
ASUS	7	1	1	3	1	2	
TCL/Alcatel	8		1		2	5	
ZTE	8	4		3	1		
Lenovo	5	3		1		1	
Blackview	4	1	1		1	1	
Hisense	4	1	1		2		
Vivo	4		1		1	2	
Tinno	2			1		2	
Mobicel	2		2				
Vodafone	2	1		1			
Other (24)	24	6	3	2	9	4	
Total (45)	**529**	**58**	**78**	**114**	**270**	**459**	**235**

Table 3: Manufacturers by Android Version. The manufacturer name is based on the `ro.product.manufacturer` system property. The number of models observed for each manufacturer is a count of all of the unique values of `ro.product.model` seen for that manufacturer and thus may include several different variants that share a common marketing name. The Android version is determined by the value of `ro.build.version.release` as reported by the device.

Because the app and script used to analyze the device for PII were intended to trigger logging, the script first retrieved the logs after first connecting to the phone and then a second time after attempting functions such as toggling the WiFi connection, initiating a BT scan, and retrieving the location. We believe it is reasonable to assume that users normally take these actions. For example, toggling the WiFi triggers a scan that will also happen when devices change locations. Since older log entries are continuously replaced, an increase in the amount of PII being detected the second time that the logs are analyzed suggests that the processes triggered by our script are what is responsible for the logging of that PII. Table 4 shows an increase in the WiFi and BT scan information reported, which suggests that the system itself is logging that information as opposed to a user-installed app.

Table 4 summarizes the PII types we searched for in our crowdsourced data and the prevalence with which they were found from the perspective of devices, manufacturers, and Android versions. The detection of some of the PII types also depends on the environment of the device during the test. For example, the BT scan results will only log information were other BT devices broadcasting at the right time. Thus, our

PII type	Devices				Manufacturers		Android Version					
	Initial Log	Final Log	Detected	Prevalence	Detected	Prevalence	≤ 7	8	9	10	11	12
Android ID	50	82	92	8%	26 of 45	58%	29%	19%	11%	12%	3%	0%
Bluetooth MAC	85	123	136	11%	27 of 45	60%	44%	14%	21%	9%	10%	6%
Bluetooth Name	724	782	836	69%	40 of 45	89%	81%	55%	82%	64%	74%	61%
Bluetooth Scan MAC	5	26	26	2%	10 of 45	22%	13%	0%	1%	1%	1%	6%
Bluetooth Scan SSID	13	26	26	2%	7 of 45	16%	9%	3%	4%	1%	1%	3%
Coarse Location	246	245	292	24%	14 of 45	31%	15%	14%	25%	23%	19%	42%
Email Address	192	180	198	16%	4 of 45	9%	57%	23%	16%	11%	28%	0%
Fine Location	224	231	272	22%	14 of 45	31%	15%	12%	25%	20%	18%	40%
IMEI	14	10	16	6%	9 of 30	30%	27%	5%	4%	-	-	-
Phone Number	11	12	14	3%	4 of 40	10%	5%	4%	3%	3%	-	-
Serial Number	43	40	49	4%	12 of 45	27%	26%	5%	4%	1%	4%	3%
WiFi Randomized MAC	242	532	539	78%	19 of 20	95%	-	-	-	-	71%	89%
WiFi Router MAC	500	795	814	67%	35 of 45	78%	54%	38%	55%	44%	76%	94%
WiFi Router SSID	597	808	829	68%	35 of 45	78%	60%	63%	61%	51%	75%	83%
WiFi Scan MAC	89	167	175	14%	19 of 45	42%	22%	15%	24%	7%	12%	23%
WiFi Scan SSID	364	462	475	39%	24 of 45	53%	26%	37%	27%	26%	37%	69%
Any PII Detected	998	1108	1142	94%	45 of 45	100%	97%	91%	95%	88%	95%	100%

Table 4: PII Count. The logs are retrieved twice: the former at the start of the script and the latter at the end. The counts in the devices section indicate the number of devices with that PII type detected during each log sampling as well as a union of the two. The prevalence is calculated based on the number of devices for which that PII type was sought (cf. §4.4). The manufacturers section indicates the number of unique manufacturers that had at least one device with identified PII in the set where we searched for that PII type. Note that as shown in Table 3, a small number of manufacturers dominate the dataset. The final section is similar but grouped by Android version.

results are a lower bound for what may occur. Note that the primary goal of this part of the experiment was to understand whether the system log personally identifies the owner.

Table 4 also demonstrates how PII logging changed among versions of Android, with some types maintaining their prevalence while others changed. The detection of Android IDs changed significantly for a couple of reasons. With older releases of Android, the Android ID was generated when the device was first configured, whereas as of Android 8, it is scoped by signing key and user (i.e., apps signed by different keys will see a different value for the Android ID) [18]. Furthermore, as of Android 10, access to non-resettable identifiers, such as MAC addresses, has also become restricted [13].

We broadly detected PII across the different manufacturers with *all* manufacturers having at least one device with at least one piece of PII detected in its logs. In Table 4, a manufacturer is considered positive if one single device has PII detected in its logs. As noted earlier, our dataset is dominated by the major manufacturers that ship the most devices. Thus, it is interesting to note the PII types that were detected by a strong majority of manufacturers even though most were represented by a small number of devices. The BT Name and WiFi Randomized MAC address were detected on devices by most manufacturers, suggesting that this information is being logged by processes that are common across all AOSP derivatives, such as by the common services discussed in §5.1 or by very popular apps. Logging of PII does not seem to be restricted to devices produced by a small group of manufacturers, but seems to be common across the ecosystem.

Leveraging the information described in §5.1 and the dataset of preinstalled apps gathered by Gamba et al. [35], we attributed the source of the PII in the log to either being from the OS, an app found to have been preinstalled previously, or a normal app; we display the results in Table 5. Since we did not collect a list of the apps that were installed on each device—which could be uniquely identifying—we relied on the reported log tags and process information available. If we could match the process name to a package name in the above dataset and that package had been observed to be preinstalled in the past, then we counted it in the preinstalled category. If the process information or log tag matched a known system process, then we classified it in the OS category. Note that the division between these two groups is somewhat arbitrary since many of the core apps that make the device usable are in the preinstalled category. Normal apps and log entries that we could not account for are grouped together in the last column. Thus, these numbers should be taken as approximate at best.

Confirming our findings in §5.1, we found that a majority of devices logged location information either explicitly as GPS coordinates or through surrogates, such as WiFi and BT scan data. Our results show that 94.1% of devices had at least one piece of PII detected in the logs that corresponds to 1,142 of 1,214 devices.

5.3 Case Studies in Logging

A manual analysis of select apps on our own devices allows us to execute apps in a more realistic manner than is possible under automated testing and to more exhaustively examine

PII Type	Attributed Code					
	OS		Preinstalled		App/Other	
Android ID	44	47%	44	47%	5	5%
Bluetooth MAC	124	66%	64	34%	1	1%
Bluetooth Name	574	44%	649	49%	91	7%
Bluetooth Scan MAC	2	7%	26	93%	0	0%
Bluetooth Scan SSID	7	21%	25	76%	1	3%
Coarse Location	14	4%	273	85%	33	10%
Email Address	140	46%	157	52%	7	2%
Fine Location	12	4%	255	87%	26	9%
IMEI	1	6%	14	82%	2	12%
Phone Number	2	13%	12	75%	2	13%
Serial Number	35	64%	19	35%	1	2%
WiFi Randomized MAC	506	44%	139	12%	508	44%
WiFi Router MAC	791	60%	339	26%	193	15%
WiFi Router SSID	807	61%	407	31%	110	8%
WiFi Scan MAC	154	81%	29	15%	6	3%
WiFi Scan SSID	445	56%	257	32%	90	11%
Any PII Detected	1075	40%	871	32%	747	28%

Table 5: Number of devices with each PII type detected. Some types are only searched for on some versions (cf. §4.4).

logging behaviour. Recall that our field study did not collect any logs from users' devices for privacy reasons. We therefore augment this with case studies of particular apps in the popular lists of categories on the Google Play Store to examine possible privacy risks.

Logging Activity Names by the OS As described in §2, the Android OS logs the names of UI-based Android app components, called *activities*, whenever one is launched. Developers, who may be unaware of this logging, often use developer-friendly names for these activities, such as "CheckoutActivity," "OrderSummaryActivity," "ResetPasswordActivity," etc. This can result in a user's actions within an app being revealed in the log by virtue of this logging.

From May–June 2022, we tested 230 Android apps primarily from the popular lists of every Google Play Store category. We examined the manifest files, which declare a list of activities, to assess whether the activity names could reveal sensitive information about a user or their interactions. When revealing activity names were encountered, we ran the app manually on a stock Google Pixel 3a running the latest version of Android 12 to see if the activity name was logged while using the app. We uncovered several examples of apps that had activity names that potentially revealed sensitive information about a user or their actions within an app.

Clue Period Tracker (`com.clue.android`, version 2937) is an ovulation and pregnancy tool that helps users who are menstruating, pregnant, or postpartum to track information about their health. We found that certain app activities are *only* visited when a user indicates that they are pregnant and so the presence of `PregnancyHomeActivity` in the logs reveals that the user had indicated to the app that they are pregnant. Furthermore, the transition out of the pregnancy mode uses a different activity depending on the reason for the transition provided by the user. A different activity is launched when a user moves directly into the postpartum mode versus when a user clicks "No Longer Pregnant"— the latter selection prompts users to select options such as abortion and miscarriage, along with a few variations of successful childbirth. Note that the selection made in the "No Longer Pregnant" pathway does not cause specific activities to be launched, but the differentiation between the standard postpartum exit and the alternative is clearly defined. Similarly, *Xiaomi Health* (`com.xiaomi.hm.health`, version 50581) has several health activities related to menstruation and pregnancy.

Apps with a calling feature, such as *Discord* (`com.discord`, version 126012) or *Microsoft Teams* (`com.microsoft.teams`, version 2022304413), also feature sensitive activity names that reveal the timing of activities. With both apps, we were able to identify when a call occurred due to their aptly named in-call activities. Microsoft Teams also had activities related to leaving a call, so the duration of a call can be calculated from the log. When Discord shows an incoming call, `WidgetVoiceCallIncoming` is logged revealing that the call was incoming as opposed to outgoing.

We also found several other apps with sensitive activity names. *Discount* (`com.ideomobile.discount`, version 2375), an Israeli banking app, has several activity names that expose various banking information, such as creating a foreign account, applying for a new loan, repaying a loan early, ordering a credit card, examining a mortgage loan, examining a pension, and freezing a credit card. *Norton 360: Mobile Security* (`com.symantec.mobilesecurity`, version 220520002) has an activity titled "MalwareFoundActivity" that static analysis showed launches when the app detects the presence of malware on a device. *Twitter* (`com.twitter.android`, version 29440001) has a "ToxicTweetNudgeActivity" referring to a reminder it uses to prompt users to review and revise potentially harmful or offensive replies. Dynamic testing showed that when a user drafts a message that Twitter considers potentially toxic, it launches this activity to prompt the poster to reconsider. Finally, 20 applications had at least one activity related to resetting or changing a password.

Apps that log PII Several of the applications that we examined log potentially sensitive data during their normal usage. *Google Calendar* (`com.google.android.calendar`, version 2017077928) is one of the preinstalled apps on many Google-certified Android smartphones. We tested the app on both a Samsung Galaxy S22 and a Google Pixel 3a and found that the email address of the calendar event *creator*—not necessarily the owner of the device—was logged whenever an event or task on that calendar was triggered.

The preinstalled *Contacts* app of the Google Pixel (`com.google.android.contacts`, version 2826706), logs metadata during the process of sharing a contact. When a contact is shared, the name of the created VCF file encodes the contact's real name, e.g., "John Smith" results in a file named "John%20Smith.vcf," and that filename is then logged.

We investigated the *Clock* app (com.google.android.deskclock, version 72004798) that is preinstalled on the Pixel 3a and found several lines in the logs that occurred only when an alarm was triggered. The message "AlarmClock: Adjusting state from [...] to FIRING" is logged whenever an alarm begins to ring. A similar message is also logged when the user dismisses the alarm by changing its state to DISMISSED by ending the alarm or to SNOOZED by snoozing the alarm. A line is also logged by "ConditionProviders.SCP" when the alarm is triggered that expresses the current time and the next alarm time, or the UNIX epoch, if there is no currently-scheduled future alarm.

Logging by SDKs We examined apps from *CVS* (com.cvs.launchers.cvs, version 778) and *Shopper's Drug Mart* (com.loblaw.shoppersdrugmart, version 706)—two prominent pharmacies in the U.S. and Canada, respectively. Both apps featured multiple pharmacy-related functionality along with an online store and COVID-related features. We found logs tagged with AdobeExperienceSDK detailing a user's actions within each app, and statically and dynamically attributed this logging to Adobe Analytics [2]. Once we saw this tag repeatedly, we more carefully looked at the apps described in §4.2 to identify others with this same log tag.

The CVS app logged the categories used to filter the store; various categories of the store page exist and when a user selects a category to refine their search, this information is present within the logs. We were able to discern from the log messages when someone navigated multiple categories to view emergency contraceptives (e.g., Plan B). The Shopper's Drug Mart app logs when a user views a product in addition to when they add it to their shopping cart. The specific product along with other user information—including the SHA-256 hash of their login email and whether they are connected to a specific local pharmacy—is included in the log messages.

We investigated this logging and discovered through network analysis that the log messages occur at the same time that this analytics information is submitted to Adobe's servers. Our investigation further revealed that the logging is controlled by a developer-set LoggingMode configuration option for the SDK and that setting it to a lower level disabled the resulting log messages. We examined 76 apps that used Adobe Analytics and found that 95% of them set the logging level to the DEBUG level or higher, resulting in these apps' analytics network traffic getting written to the logs. Moreover, we found that AdobeExperienceSDK was the log tag used for logging the Advertising ID, Android ID, router SSID, and precise GPS coordinates. Note that, according to the SDK documentation, the default log level for this library is ERROR, which is reserved for the most serious errors (i.e., 95% of the apps we found using the Adobe Analytics SDK had changed the default logging level). Furthermore, the documentation clearly warns that "using DEBUG or VERBOSE log levels may cause performance or security concerns" [3]. Yet, Adobe's "Getting Started Guide" features an example in which the log

```
12 public class MainApp extends Application {
13   ...
14   @Override
15   public void on Create(){
16     super.onCreate();
17     MobileCore.setApplication(this);
18         MobileCore.setLogLevel(LoggingMode.DEBUG);
19     ...
```

Figure 5: The Adobe "Getting Started Guide" [4] features a code snippet in which the log level is changed to DEBUG.

level is set to DEBUG (Figure 5) [4]. (We hypothesize that most developers using this SDK are copying and pasting from the documentation, while ignoring the advice to change the log level in production.) This suggests that at least some of the logging done by embedded SDKs is a result of the app developer misusing the SDK and not configuring it appropriately when releasing the production version of an app.

6 Results: Where Do Logs Go?

In this section we consider who has access to the information in the system log files. Using a taint analysis tool, we identified several instances of preloaded system apps that access and leak system logs to the cloud or write them to the shared storage, potentially exposing system logs and any PII contained in them to any user-installed app with the READ_EXTERNAL_STORAGE permission. Next, we examined OEM privacy policies to identify actors explicitly acknowledging the collection of log data.

6.1 Accessing and Leaking System Logs

Using static analysis methods, we analyzed the preinstalled apps dataset gathered by Gamba et al. [35] using crowdsourcing mechanisms [44] to assess the prevalence of log collection on Android and which parties may access those logs. As of October 8, 2022, this dataset contained 1,395,271 apps from 36,061 device models from 1,069 vendors.

In total, we identified 149,622 apps (1,915 unique package names) requesting the READ_LOGS permission in their manifest file. We note that the READ_LOGS logs permission has a protection level of signature|privileged, meaning that it can only be acquired by system apps either signed with the same certificate as the system or explicitly listed by the device manufacturer [10]. The majority of the apps found reading logs are indeed OEM-developed apps signed with OEM certificates. Unfortunately, in contrast to apps available on app markets, preinstalled apps lack metadata that could be used to categorize them by their purpose or functionality, and the use of self-signed certificates impedes the accurate identification of the developer [35, 42]. To overcome this limitation, we

Package	# OEMs
com.google.android.gsf	745
com.google.android.feedback	705
com.google.android.gsf.login	658
com.google.android.gms	584
com.mediatek.mtklogger	253
com.baidu.map.location	99
com.google.android.googlequicksearchbox	85
com.adups.fota	64
com.kingroot.kinguser	63
com.mediatek.duraspeed	42
com.debug.loggerui	41
com.google.android.projection.gearhead	34
com.gangyun.beautysnap	30
com.qualcomm.qti.perfdump	27
com.mediatek.mobilelog	21
com.redstone.ota.ui	20
com.UCMobile.intl	20
com.qualcomm.qti.RIDL	19
com.wsandroid.suite	19
com.cleanmaster.mguard	19
com.mediatek.engineermode	18
com.rock.gota	17
com.sprd.runtime	16
com.lookout	15
com.softwinner.fireplayer	15
com.speedymovil.wire	15
com.huaqin.runtime	14
com.evernote	14
com.qualcomm.logkit	13
com.sprint.ms.smf.services	12
com.amazon.mp3	12
com.gionee.systemmanager	12
com.google.android.apps.turbo	12
com.verizon.obdm	12
com.amazon.venezia	11
com.baidu.browser.inter	11
com.bbm	11
com.gionee.amisystem	11
com.gionee.softmanager	11
com.iflytek.speechsuite	11
com.tmobile.pr.adapt	11

Table 6: Preloaded apps requesting the READ_LOGS permission found on devices from at least 10 different OEMs.

relied on the package name and certificate of the app to infer the developer and nature of the app. This allows us to not only find instances of critical Android components accessing logs, but also non-OEM related apps present on devices from a large variety of vendors and brands.

Table 6 lists a subset of preinstalled apps requesting access to system logs found across more than 10 different OEMs. The most common ones were from Google (e.g., Google Play and GMS), and are present on every Google-certified device. For clarity, we exclude core Android components such as `com.android.contacts`. The data suggests that OEMs may customize the open-source version of these core components to add their own functionality, including collecting system logs. This is the case, for example, for the package `com.android.ActivityNetwork` that is present on certain Lenovo, Motorola, TCT, Alps, and Acer mobile phones.

We observed third-party non-OEM preinstalled apps request access to READ_LOGS. Examples included apps developed and signed by MNOs like Verizon, AT&T, Vodafone, Telefonica, MetroPCS, or Sprint; large companies like Amazon, Baidu, Microsoft, Yahoo!, Qihoo360, Tencent, and Evernote; browsers already known for implementing privacy-intrusive behaviours (e.g., versions of UC Mobile and Baidu Browser); utility apps such as device cleaners (e.g., CleanMaster's MGuard), parental control apps (e.g., Kidoz), and anti-virus software (e.g., Lookout and McAfee); Viber's VoIP client; and even preinstalled malware on low-end Android devices (e.g., `com.rock.gota`) [56]. In addition, apps from companies offering Firmware Over-the-Air (FOTA) components like Redstone and Adups, which are known for distributing and installing malware [21], also request access to the logs. While some of these package names are available on Google Play, the versions found on preinstalled devices differ.

Although we cannot confidently conclude how these apps were installed on the system partition, it is possible that some of these apps may abuse the lack of control over privileged FOTA components to gain access [21]. In fact, only when these apps are installed on the system partition can they effectively access the logs. Therefore, if Android's policies are interpreted strictly [14], these apps would be considered as third-party apps and consequently barred from accessing the READ_LOGS permission because *"log entries can contain the user's private information."*

Do these apps leak logs data? Dynamically testing whether system apps requesting the READ_LOGS permission upload logs to servers is not trivial. Preinstalled apps may use features such as the shared user ID [11][3] and have native dependencies that may cause errors when installed in a testing environment. To overcome these limitations and scale-up the analysis, we built a static taint analyzer on top of Androguard [6] to identify which preinstalled (i.e., system) applications read the system logs and then leak it to different sinks such as network sockets or to files on the SD card which consequently leaks this data to any application with access to the storage permission. Our tool enriches the analysis object created by Androguard [6] to add extra cross-references to account for asynchronous communications such as intents. Unfortunately, static analysis tools are not effective at identifying hidden behaviours that rely on native code, dynamic DEX code loading, or reflection, and may also render false positives due to legacy or dead code that is never executed [35].

To focus our analysis and contextualize it with the results presented in the previous section, we cross-referenced the build fingerprints of the devices observed in our crowdsourcing campaign (§5.2) with the database of preinstalled apps collected using Firmware Scanner [44]. Of the 772 unique build fingerprints, 315 were also present in the Firmware Scan-

[3] If an app declares a shared user ID that is already in use by another app on the system, then the installation will fail.

ner database and were associated with 1,319 apps requesting the READ_LOGS permission on those devices. When grouped by their package name and signing certificate to identify the developer or the party responsible for the app, we found 237 groups of apps. For each of these groups, we manually inspected the code of the most recent version and found 63 apps that run `logcat` as a shell command. We found that 7 of the 63 filter the logs after retrieving them, but the rest retain the entire log. For example, some apps search for a specific pattern, such as a package name or PID, or are triggered upon specific events, like an app crash. Surprisingly, 15 of these apps implemented code to save the raw logs directly to the SD card. This may allow actors that request the storage permission to read and subsequently upload such logs files from public storage and thus bypass the READ_LOGS permission entirely. More worrying, we found 9 apps that post raw logs to the Internet. In one case, the logs are sent to a Firebase instance—a third-party service operated by Google. In total, 4,598 users in the preinstalled dataset had at least one app in the system partition either saving the raw logs to the SD card or uploading them to the cloud.

Case studies Log leaks can be triggered by specific events, such as app crashes. This is the case with the Google Feedback app (`com.android.feedback`), which allows the user to attach the system logs to the bug report submitted to Google. We confirmed that the entire system log is sent in an HTTP POST request to `www.google.com/tools/feedback/android/__submit`, by crashing an app and triggering this feedback operation while monitoring the network traffic. Not only were the logs from the crashing app sent, but also the logs from the operating system and the logs of *all* other user-space apps running on the phone—including those unrelated to the crash event. In other cases, log collection and leakage may be triggered by the reception of an intent with a specific action, which hints at the existence of one or more additional apps that must have the logic to trigger said behaviour. As previously described, the majority of these preinstalled apps were signed by the manufacturer of the device. However, some of these apps were signed by third-party companies (e.g., Vodafone). We also found other cases where the signing certificate did not give useful information about the identity of the company behind it (e.g., by using names such as `C=IL, ST=il, L=TLV, O=Central antivirus, OU=antivirus, CN=Dror Shalev` or by using debug certificates). This confirms the lack of control over the software supply chain and highlights the attribution challenges for preinstalled apps, as has already been pointed out in prior work [35].

One interesting case is the set of apps developed by Mobile Posse [29], an advertising company bought by Digital Turbine in February 2020 [30]. We found 8 such apps scattered across 68 unique devices from well-known manufacturers. Manual code inspection showed that all of these apps have the same code to access the logs. First, the code explicitly checks if the app was granted the READ_LOGS permissions and, if so, runs `logcat` with the `-d` option[4] and saves the output. It appears that the logs are then converted into a JSON-formatted string and sent as an HTTP POST request to the cloud. While we were not able to confirm with complete certainty the destination of this request, we found strong indications in the code that the logs are sent to a machine hosted on AWS. The app also contains a JSON-formatted string called "schedule" that appears to contain data collection instructions, including which components are to be collected and at what frequency. The schedule contains, among other things, the `collect_system_log_schedule` operation, which gives the `logcat` command to run: "`logcat -v time -d *:e`." The other operations seem to instruct the apps to collect sensitive information, such as the list of installed apps, app usage, visited URLs, geographic and cell location, call history, signal strength, network info, connection speed (with links to test upload and download speed), boot time, SMS usage, battery status, and memory usage. This information would then be sent alongside the logs to the aforementioned domain.

6.2 Privacy Policy Analysis

As described earlier, under the current Android security model it is not possible for an app to obtain the required READ_LOGS permission that allows it to read the entire system log unless it is preinstalled—this can only be done by the OEM or another supply chain actor that partners with the OEM. Thus, it is informative to examine the privacy policies of major manufactures to reveal what claims are made about the log information that they say that they collect, since any app with the READ_LOGS permission can only have been included on the device by their action or consent.

We manually examined the published privacy policies of some of the OEMs with the largest market share and took note about what log information they claimed to collect. A summary of the log information automatically collected is listed in Table 7. The privacy policies of some of the largest OEMs include language that covers their collection of device identifiers, log data, and crash reports—which are expected to include log data. The privacy policies published by Oppo and OnePlus—which share much of the same text for their policies (both of them are manufactured by BBK electronics)—go further than most and clearly show that phone vendors understand what information can end up in the system log:

> *Log information:* [Log data] may sometimes include your personal data, such as phone number, email address, Google account or Facebook account. However, we have implemented security measures to ensure that this information is used only for error log analysis and not for personal identification or other purposes. [49, 50]

[4] The `-d` option makes `logcat` dump the entire log buffer and exit afterwards instead of waiting for more entries to arrive.

Manufacturer	Log Data Collected
Google [38]	"includes things like your device type and carrier name, crash reports"
Huawei [43]	"event information (such as errors, crashes, restarts, and updates)"
OnePlus [49]	"error or crash log will contain information collected at the time of the event"
Oppo [50]	"error or crash log will contain information collected at the time of the event"
Samsung [54]	"diagnostic, technical, error, and usage information"
Sony [58]	"error related data and configuration, functionality, and performance data"
Vivo [65]	"product interactions, crash records, and diagnostic data"
Xiaomi [68]	"temporary message history, standard system logs, crash information, log information generated by using the service"
ZTE [71]	"device data, software data, and service log data"

Table 7: Types of log information collected according to the privacy policies of several device manufacturers.

Furthermore, device manufacturers, MNOs, and other OEM partners may preinstall third-party software in addition to their own. As noted in §1, Gamba et al. described a complex supply chain of providers whose software is often preinstalled for various business reasons [35]. Included analytics services, such as Mobile Posse [29], which was acquired by Digital Turbine [30], have privacy policies that indicate that they may collect log information [32]. Furthermore, in their notice that is required under California law [31], they note that their mobile delivery platform may be installed by default. As described in §6.1, they follow through with this collection.

7 Conclusions

Logging remains an effective and efficient way to debug software throughout the development cycle. As the trend of "debugging in deployment" continues, so will the collection and transmission of system logs. Thoughtful logging allows developers to quickly reconstruct errors that occur without the burden of replicating issues in a debugger. The logging model on Android, however, is more complicated. Any app can conveniently use the built-in logging framework, and the absence of a console, the Android system log functions as a form of *standard output*, thus amplifying concerns about third-party collection of these logs after deployment. Indeed, the flag that controls `wpa_supplicant`'s debug logging was called `CONFIG_NO_STDOUT_DEBUG`. This situation is akin to both a personal computer manufacturer and the owner's Internet provider having access to the system logs of the computer and any routine logging from any programs that run on it.

Our results, both in the lab and in the wild, show that logging sensitive data, including identifiers, location and proximate data, is prevalent across phone models. This behaviour can be traced back not only to user-installed apps, but also to system components and preinstalled apps. Thus, it is important that developers, including those of Android itself, take Google's admonishment to developers that they *not log private data*. It is more important, however, that access to these logs is put in the control of the device owner, not the agglomeration of corporate interests that play some role in the phone's manufacturing or connectivity. Ideally, the platform is stable enough so that the manufacturer does not need to continue being the device administrator once it is in the hands of a consumer. As observed by Rosenberg [53], it is futile to sanitize thousands of logging lines and it ultimately reduces the utility of that logging. This futility is amplified by app developers who freely log arbitrary information or misconfigure SDKs.

Another issue is that of crash reporting frameworks, such as Crashlytics. Such SDKs are prevalent in apps and provide a valuable service to developers. Instead of addressing software issues in development, app makers can quickly release buggy code to real users and debug in deployment by having logs sent to a third party for analytics. In earlier times, users of such developmental code were called "beta testers." Developers who enlist all their users as beta testers are more agile with deploying features and fixing issues than those who do not, meaning that developers who respect users' privacy have a harder time competing. We need to foster the understanding—either among developers or lawmakers—that it is *unreasonable* to expect that writing a piece of software entitles you to indefinitely access detailed information regarding precisely how it is used. Our advice for developers is to (*i*) disable logging when preparing a release build; (*ii*) assess what information activity names can reveal, assuming that they are logged; and (*iii*) inspect the logging output while running a release candidate to ensure that any included SDKs are correctly configured as it pertains to their logging.

We responsibly disclosed our findings to Google. We have engaged with them in conversations about improving the logging framework. In response to our disclosure, Google introduced a new control in Android 13 and above to help users manage access to device logs, with the goal of preventing entities from surreptitiously collecting user logs as a matter of routine [37]. (We also disclosed our results to a few OEMs, who denied collecting any logging data or ignored us.)

Acknowledgements

We are deeply grateful to our field study participants who provided us with valuable data. We thank our anonymous reviewers and shepherd for their constructive feedback, the Canadian Digital Service for deploying a custom version of the COVID-19 alert app for testing, and our contacts at Google who clarified some of our questions. We also thank Vinuri Bandara (IMDEA Networks) and Eduardo Blazquez (UC3M) for their help in the analysis of preinstalled applications.

This research was supported by the Spanish Government grant ODIO (PID2019-111429RB-C21 and PID2019-111429RBC22); the Region of Madrid, co-financed by European Structural Funds ESF and FEDER Funds, grant CYNAMON-CM (P2018/TCS-4566); and by the EU H2020 grant TRUST aWARE (101021377). Joel Reardon was supported by the Cisco University Research Program Fund and an NSERC Discovery Grant. Narseo Vallina-Rodriguez was supported by the project REACT-CONTACT-CM-23479, funded by Comunidad de Madrid and the European Regional Development Fund, and by a Ramon y Cajal Fellowship (RYC2020-030316-I). Serge Egelman was supported by the U.S. National Science Foundation (CNS-1817248) and the National Security Agency (H98230-18-D-0006).

The opinions, findings, and conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of any of the funding bodies.

References

[1] Jagdish Prasad Achara, Mathieu Cunche, Vincent Roca, and Aurélien Francillon. Short paper: WifiLeaks: Underestimated privacy implications of the access_wifi_state android permission. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks - WiSec '14*, pages 231–236, Oxford, United Kingdom, 2014. ACM Press.

[2] Adobe. Adobe analytics: Web analytics for better business intelligence. https://business.adobe.com/products/analytics/adobe-analytics.html. Accessed 2022-Oct-11.

[3] Adobe. Debugging & lifecycle metrics. https://developer.adobe.com/client-sdks/documentation/getting-started/enable-debug-logging/, 2023.

[4] Adobe. Get the Experience Platform SDK. https://web.archive.org/web/20230127083605/https://developer.adobe.com/client-sdks/documentation/getting-started/get-the-sdk/, 2023. Accessed: February 28, 2023.

[5] Jaber Al Nahian. How to easily change or fake your android device model and brand name. TechGainer, July 14 2014. https://www.techgainer.com/change-fake-android-device-model-number-and-brand-name/.

[6] Androguard Team. Androguard. https://github.com/androguard/androguard/. Accessed 2022-Sep-29.

[7] Android Open Source Project. Security tips. https://developer.android.com/training/articles/security-tips.

[8] Android Open Source Project. Adding a new device. https://source.android.com/setup/develop/new-device#build-variants, 2021.

[9] Android Open Source Project. Logcat command-line tool. https://developer.android.com/studio/command-line/logcat, 2021.

[10] Android Open Source Project. Privileged permission allowlisting. https://source.android.com/devices/tech/config/perms-allowlist, 2021.

[11] Android Open Source Project. <manifest>. https://developer.android.com/guide/topics/manifest/manifest-element, 2021.

[12] Android Open Source Project. Android debug bridge (adb). https://developer.android.com/studio/command-line/adb, 2022.

[13] Android Open Source Project. Best practices for unique identifiers. https://developer.android.com/training/articles/user-data-ids, 2022.

[14] Android Open Source Project. Manifest.permission. https://developer.android.com/reference/android/Manifest.permission#READ_LOGS, 2022.

[15] Android Open Source Project. Meet android studio. https://developer.android.com/studio/intro, 2022. Accessed 2022-Oct-04.

[16] Android Open Source Project. Privacy changes in android 10. https://developer.android.com/about/versions/10/privacy/changes#location-telephony-bluetooth-wifi, 2022. Accessed 2022-Sep-22.

[17] Android Open Source Project. Publish your app. https://developer.android.com/studio/publish, 2022. Accessed 2022-Oct-03.

[18] Android Open Source Project. Settings.secure. https://developer.android.com/reference/android/provider/Settings.Secure, 2022. Accessed 2022-Sep-22.

[19] Android Open Source Project. Signing builds for release. https://source.android.com/devices/tech/ota/sign_builds, 2022.

[20] Android Open Source Project. Understanding logging. https://source.android.com/devices/tech/debug/understanding-logging, 2022.

[21] Eduardo Blazquez, Sergio Pastrana, Alvaro Feal, Julien Gamba, Platon Kotzias, Narseo Vallina-Rodriguez, and Juan Tapiador. Trouble over-the-air: An analysis of FOTA apps in the android ecosystem. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1606–1622, San Francisco, CA, USA, May 2021. IEEE.

[22] C. Scott Brown. Here are the phone update policies from every major android manufacturer. https://www.androidauthority.com/phone-update-policies-1658633/, July 2 2022. Accessed 2023-Feb-16.

[23] Simon Chan. Android debug bridge (ADB) for web browsers. https://github.com/yume-chan/ya-webadb, 2022.

[24] CVE-2017-9615. https://www.cve.org/CVERecord?id=CVE-2017-9615, June 2017.

[25] CVE-2018-1999036. https://www.cve.org/CVERecord?id=CVE-2018-1999036, July 2018.

[26] CVE-2021-31815. https://www.cve.org/CVERecord?id=CVE-2021-31815, April 2021.

[27] CWE-532: Insertion of sensitive information into log file. https://cwe.mitre.org/data/definitions/532.html, July 2006.

[28] Yves-Alexandre de Montjoye, César A. Hidalgo, Michel Verleysen, and Vincent D. Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific Reports*, 3(1):1376, December 2013.

[29] Digital Turbine – Mobile Posse. https://mobileposse.com/. Accessed 2022-Jun-07.

[30] Mobile Posse acquired by Digital Turbine. https://www.crunchbase.com/acquisition/mandalay-digital-group-acquires-mobile-posse--1e380e32. Accessed 2022-Jun-07.

[31] Digital Turbine. California privacy policy. https://www.digitalturbine.com/california-privacy/, 2020. Accessed 2022-Sep-08.

[32] Digital Turbine. Privacy policy. https://www.digitalturbine.com/privacy-policy/, 2022. Accessed 2022-Sep-08.

[33] F-Droid Limited and Contributors. F-Droid - free and open source android app repository. https://f-droid.org/. Accessed 2022-Sep-08.

[34] Alvaro Feal, Julien Gamba, Juan Tapiador, Primal Wijesekera, Joel Reardon, Serge Egelman, and Narseo Vallina-Rodriguez. Don't accept candy from strangers: An analysis of third-party mobile SDKs. In *Data Protection and Privacy: Data Protection and Artificial Intelligence*, volume 13 of *Computers, Privacy, and Data Protection*, page 1. Gordonsville, 2021.

[35] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. *IEEE Symposium on Security and Privacy (SP)*, 2020.

[36] R. Gerhards. The syslog protocol. Technical Report RFC 5424, RFC Editor, March 2009. https://doi.org/10.17487/rfc5424.

[37] Google. Manage your device logs on android. https://support.google.com/android/answer/12986432.

[38] Google Inc. Privacy policy – privacy & terms – Google. https://policies.google.com/privacy, 2022. Accessed 2022-May-26.

[39] Reilly Grant, Ken Rockot, and Ovidio Ruiz-Henríquez. WebUSB API. https://wicg.github.io/webusb/, 2022.

[40] Dianne Hackborn. READ_LOGS permission is not granted to 3rd party applications in jelly bean (api 16). https://groups.google.com/g/android-developers/c/6U4A5irWang/m/AvZsrTdfICIJ, Jul 2012.

[41] Dianne Hackborn. READ_LOGS permission is not granted to 3rd party applications in jelly bean (api 16). https://groups.google.com/g/android-developers/c/6U4A5irWang/m/dEsqi0dyPkkJ, Jul 2012.

[42] Kaspar Hageman, Álvaro Feal, Julien Gamba, Aniketh Girish, Jakob Bleier, Martina Lindorfer, Juan Tapiador, and Narseo Vallina-Rodriguez. Mixed signals: Analyzing software attribution challenges in the android ecosystem. *IEEE Transactions on Software Engineering*, pages 1–16, 2023.

[43] Huawei Technologies Co. Ltd. Privacy statement - HUAWEI Global. https://consumer.huawei.com/en/privacy/privacy-policy/, 2019. Accessed 2022-May-26.

[44] Internet Analytics Group - IMDEA Networks. Firmware Scanner. https://play.google.com/store/apps/details?id=org.imdea.networks.iag.preinstalleduploader, 2020. Accessed 2022-Sep-29.

[45] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1999.

[46] C. Lonvick. The BSD syslog protocol. Technical Report RFC 3164, RFC Editor, August 2001. https://doi.org/10.17487/rfc3164.

[47] Alfred Ng. Google promised its contact tracing app was completely private—but it wasn't. https://themarkup.org/privacy/2021/04/27/google-promised-its-contact-tracing-app-was-completely-private-but-it-wasnt, April 27 2021. Accessed 2023-Feb-16.

[48] John O'Brien and Kimmo Lehtonen. Counterfeit mobile devices - the duck test. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 144–151, Fajardo, October 2015. IEEE.

[49] OnePlus. Privacy policy - OnePlus (Global). https://www.oneplus.com/global/legal/privacy-policy, 2021. Accessed 2022-Sep-12.

[50] OPPO. Privacy policy. https://www.oppo.com/en/privacy/, 2021. Accessed 2022-Sep-12.

[51] Joel Reardon. Why google should stop logging contact-tracing data. https://blog.appcensus.io/2021/04/27/why-google-should-stop-logging-contact-tracing-data/, 2021.

[52] Regents of the University of California. What needs CPHS/OPHS review. https://cphs.berkeley.edu/review.html, 2023. Accessed 2023-Jan-30.

[53] Dan Rosenberg. [patch v3] restrict unprivileged access to kernel syslog. https://lwn.net/Articles/414813/, Nov 2010.

[54] Samsung Electronics Co., Ltd. Samsung privacy policy. https://privacy.samsung.com/policy/samsung, 2020. Accessed 2022-May-26.

[55] Piotr Sapiezynski, Arkadiusz Stopczynski, Radu Gatej, and Sune Lehmann. Tracking human mobility using WiFi signals. *PLOS ONE*, 10(7):e0130824, July 2015.

[56] Secure-D Lab. com.rock.gota. https://lab.secure-d.io/com-rock-gota/, 2018.

[57] Ashkan Soltani. Privacy trade-offs in retail tracking. https://www.ftc.gov/policy/advocacy-research/tech-at-ftc/2015/04/privacy-trade-offs-retail-tracking, Apr 2015. Accessed 2022-Aug-23.

[58] Sony of Canada Ltd. Privacy code. https://corporate.sony.ca/view/privacy.htm, 2020. Accessed 2022-May-26.

[59] Statista. Global smartphone market share from 4th quarter 2009 to 2nd quarter 2022. https://www.statista.com/statistics/271496/global-market-share-held-by-smartphone-vendors-since-4th-quarter-2009/, July 2022. Accessed 2022-Sep-22.

[60] Robert Triggs. How to identify a fake or counterfeit smartphone. https://www.androidauthority.com/spot-fake-phone-882017/, June 2022. Accessed 2022-Oct-11.

[61] U.S. Federal Trade Commission. Mobile advertising network InMobi settles FTC charges it tracked hundreds of millions of consumers' locations without permission. https://www.ftc.gov/news-events/news/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked-hundreds-millions-consumers, Jun 2016. Accessed 2022-Aug-23.

[62] U.S. Federal Trade Commission. Advertising platform OpenX will pay $2 million for collecting personal information from children in violation of children's privacy law. https://www.ftc.gov/news-events/news/press-releases/2021/12/advertising-platform-openx-will-pay-2-million-collecting-personal-information-children-violation, Dec 2021. Accessed 2022-Aug-23.

[63] U.S. Federal Trade Commission. USA v. OpenX Technologies, Inc., a Delaware corporation. https://www.ftc.gov/system/files/documents/cases/ecf_3-1_-_stipulated_order.pdf, 2021.

[64] Narseo Vallina-Rodriguez, Jon Crowcroft, Alessandro Finamore, Yan Grunenberger, and Konstantina Papagiannaki. When assistance becomes dependence: characterizing the costs and inefficiencies of A-GPS. *ACM SIGMOBILE Mobile Computing and Communications Review*, 17(4):3–14, 2013.

[65] vivo Mobile Communication Co., Ltd. Privacy policy. https://www.vivo.com/en/about-vivo/privacy-policy, 2022. Accessed 2022-Sep-12.

[66] WiGLE.net. WiGLE: Wireless network mapping. https://wigle.net/. Accessed 2022-Sep-29.

[67] Ryszard Wiśniewski and Connor Tumbleson. Apktool–a tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/. Accessed 2022-Jun-02.

[68] Xiaomi Singapore Pte. Ltd. Xiaomi Privacy Policy. https://privacy.mi.com/all/en_US/, 2021. Accessed 2022-May-26.

[69] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun (Peter) Chen. Studying the characteristics of logging practices in mobile apps: A case study on F-Droid. *Empirical Software Engineering*, 24(6):3394–3434, December 2019.

[70] 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, London, ON, Canada, February 2020. IEEE.

[71] ZTE Corporation. Privacy policy. https://www.zte.com.cn/global/Privacy-Policy, 2022. Accessed 2022-Sep-12.