Panoptes: Detecting Out-of-Sight Privacy Exposure in Android

Sajjad Pourali
Concordia University, Montreal, Canada
sajjad@pourali.com

Mohammad Mannan
Concordia University, Montreal, Canada
m.mannan@concordia.ca

Abstract

Android apps frequently collect and transmit sensitive user information, even when not actively engaged by users. While existing research has extensively explored privacy concerns related to foreground app usage, the risks associated with apps operating out-of-sight, when they are minimized or closed, remain largely unexplored. We design and implement Panoptes, a dynamic analysis tool that detects and attributes data exposure in Android out-of-sight app operations at scale. By systematically triggering and monitoring background works, we conduct the first large-scale measurement of what data apps collect and how they expose sensitive information without user interaction, and attribute the execution of background works to the criteria that trigger them. We evaluated 15,456 popular apps from AndroidRank and found that 13,068/15,456 (84.5%) apps establish network connections while not visible on the device. Of these, 7534/13,068 (57.7%) continue their activity regardless of whether the app is visibly open, closed, or even if it has never been opened. Our findings also uncovered undocumented features that enable apps to collect data even when they are apparently closed. This study sheds light on the hidden privacy risks in Android, and highlights the need for developing more robust techniques to mitigate surreptitious data exfiltration through invisible activities.

1 Introduction

A significant number of Android apps extensively collect sensitive information from users and share it with multiple entities, including app servers themselves and third-party advertising/tracking companies. Existing work has extensively analyzed the implications of such data collection and sharing (e.g., [26, 27, 49, 51]); yet, these studies primarily focus on apps that expose information while actively running and visible on the phone's foreground screen. In practice, when users stop interaction, mobile apps generally remain open (in a minimized/invisible state where the app is open but not on the foreground window), or partly operational in the background (via several background works mechanisms where the app uses special APIs and is not visible to the user) even after being closed (e.g., by swiping up from the screen bottom, hold, then swiping up on the selected app). Such a design is essential for proper functionality of many apps (e.g., showing new message notification). Both Google and Apple confirmed long ago (in 2016, see e.g., [58]) that closing an app could worsen battery life, and thus mobile OSes generally keep the apps open in the background. Stopping an app from working in the background requires changes in the OS settings.

Similar to their visible state, apps can expose personal/device information while operating in the background—even if an app is used very infrequently (or not at all, post-installation). Such exposure has not been comprehensively studied so far although privacy risks from background data transmission (specifically, very aggressive location harvesting in the background) by very popular apps made the news in the past; see e.g., Uber [55], Facebook [57], TimHortons [32]. On the positive side, Google has implemented measures [2] in recent Android versions to limit such exposure.

In this paper, we focus primarily on the exposures that occur when an app is *out-of-sight* of the user, which includes scenarios such as an app is minimized (i.e., open but not visible in the UI), an app is closed/inactive, or an app is installed but never opened by the user. We introduce `Panoptes`,[1] the first Android dynamic analysis tool that can comprehensively trigger various Android background works,[2] and measure privacy exposures from them at scale.

However, analyzing background works presents unique challenges. Unlike foreground activities, which have visible UI entry points and are primarily controlled by the user, background work initiation and execution are often managed by the OS, based on specific criteria and device conditions. Finding these trigger criteria and related parameters is critical for measuring possible exposure in an experimental setting, as such conditions in reality may take long to occur naturally (e.g., change of timezone). However, finding them at scale poses a significant challenge due to the lack of comprehensive documentation and variability in app behaviors/implementations. Moreover, Android developers frequently use asynchronous APIs such as threads, Handlers, and coroutines for network operations. Since asynchronous code typically executes in different threads or tasks than its originating code, attributing network communications to specific parts of the app, including background tasks, becomes challenging. We address these challenges by systematically stimulating background works, monitoring network communications, and attributing sensitive data transmissions to specific background tasks. By focusing on out-of-sight behaviors, Panoptes provides a comprehensive view of privacy exposures that may occur without user interactions or awareness overtime.

Panoptes systematically identifies background components that can operate when the app is not open by leveraging various Android background works. Panoptes then collects criteria for their execution when specified, obtains extra values or data required for some APIs (e.g., intents used in broadcasts or services), expedites their execution, and attributes their network traffic. This allows it to determine whether an app can connect to the internet while not

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
Proceedings on Privacy Enhancing Technologies 2026(4), 358–375
© 2026 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2026-0125

[1] In Greek mythology, Argos Panoptes is a many-eyed, all-seeing giant who served as a watchman for goddess Hera: https://en.wikipedia.org/wiki/Argus_Panoptes.
[2] We consider operations performed using Android components such as Services, JobScheduler, WorkManager, BroadcastReceiver, and AlarmManager, as background works. For an overview of these components, see Sec. 1. Additionally, we define any network connections that were established when an app was not visibly open to the user, including via background works, as *out-of-sight*.

open based on defined criteria, such as changes in network connectivity or device boot events, and to report sensitive information transmitted to their servers and third-parties.

Another key contribution of Panoptes is the injection of meaningful mock records into stack traces to trace asynchronous operations at runtime. In contrast to past work on static analysis or app repacking [26, 59], our approach dynamically injects meaningful records into the stack traces of asynchronous tasks during their registration, enabling accurate tracing of data flows back to their originating components. This method remains functional even when tasks are distributed across multiple threads or triggered by system-level events, all without compromising the app's integrity.

To capture system-level activities and application-level behaviors, including the registration and execution of background tasks, we implement our instrumentation tool using eBPF [46] in conjunction with the Aya library [43] for kernel-level instrumentation, and the LSPosed [47] framework for application-level API hooking. This combination provides comprehensive dual-layer monitoring, covering both JVM and native code. Additionally, we developed techniques to trace various Android asynchronous operations and attribute network communications to their originating background tasks, thereby overcoming the technical challenges associated with monitoring non-deterministic behaviors.

We implemented Panoptes with ≈14,607 lines of code in Rust, Kotlin, and Python, and evaluated it on 15,456 popular Android apps across various app categories. Our analysis uncovered numerous instances of privacy exposure that may occur exclusively during background execution. Many apps misuse background works to persistently collect sensitive information, such as device identifiers, location data, and call history, transmitting it to third parties without user awareness. Notably, some apps established network connections and transmitted data immediately upon device boot or network connectivity changes, without any prior user interaction. Note that there is no uncertainty about our reported privacy exposure instances, except that in a real usage case, such an exposure may take long before manifestation, depending on the triggering conditions for the background works. We also uncovered undocumented Android features related to background works, which malicious apps can exploit for privacy exposure. Our main contributions and findings are summarized below.

(1) **New privacy risk measurement:** We highlight the underexplored issue of privacy exposure through out-of-sight operations in Android apps (e.g., background works), including the states when apps are minimized/inactive. We design and implement Panoptes, the first dynamic analysis framework to detect and attribute such exposures to specific background events at scale.
(2) **Dynamic asynchronous API attribution:** We present a generic, dynamic attribution technique that traces asynchronous APIs and maps asynchronous tasks to their original execution contexts at runtime without requiring modifications to the app/OS.
(3) **Novel instrumentation techniques:** We develop methods for instrumenting Android apps at both kernel and application levels, achieving approximately five times faster analysis compared to sequential analysis of single apps by enabling simultaneous multi-app tracing of asynchronous and background executions without affecting functionality.
(4) **Empirical evaluation:** To measure real-world privacy exposure from out-of-sight operations, we conduct an extensive empirical study on 15,456 Android apps. Panoptes identified that 13,884/15,456 (89.83%) apps established connections during our analysis. Of these, 13,038/13,884 (93.91%) established the connections while the apps are minimized, and 7534/13,884 (54.26%) utilized background work components, enabling execution even when the apps are closed/inactive. In total, 13,068/13,884 (94.12%) apps establish outside connections while being out-of-sight.
(5) **Privacy exposure:** We identify that 10,524/13,068 (80.5%) of the apps with out-of-sight connections transmitted privacy-sensitive information, commonly used for user tracking (e.g., GPS coordinates, Bootloader, router SSID, list of the installed apps). This corresponds to 9056/13,038 (69.5%) when apps were minimized and 7238/7534 (96.1%) when utilizing background works, with 4211/7534 (55.9%) apps exclusively sending sensitive data via background works (when app is not open).
(6) **Undocumented Android behaviors:** We identified undocumented and inconsistent Android behaviors that can facilitate apps in maintaining their execution in the background. We found 13 undocumented implicit broadcast exceptions (*static broadcasts*) (used by 3772 apps) that are frequently triggered in real-world scenarios, and as such, can be exploited as reliable entry points when an app is closed. Also certain app permissions, e.g., access to notifications, physical activity, and nearby devices, are not being revoked after a period of inactivity (3 months) by Android's hibernation feature as documented [4]. As a result, even when users stop interacting with an app for months, it can still access data in the background. We found that Android versions up to 16 are susceptible to this issue, specifically in Android 13,[3] where apps can receive intents and execute themselves without ever being opened post-installation.
(7) **Tool and dataset:** We will open-source Panoptes. To increase the analysis code coverage, we collected extra values and data for 21,907 unique intents by monitoring the apps intent usage, which we will also publish to facilitate future work.

Our tool is open-source at: github.com/Madiba-Research/Panoptes.

Ethics considerations and disclosure. For our experiments, we exclusively utilized only test Google accounts generated specifically for this research, without involving any real users. The mobile devices used in the experiments were also non-personal (used only in a lab setting). We also disclosed our findings to Google on Nov 13, 2024, including the Android 13 post-installation state issue, undocumented implicit broadcast exceptions, and improper hibernation. Google classified these as medium-severity vulnerabilities on Jan 23, 2025, rewarded our submission on Feb 13, 2025, and confirmed fixes for an upcoming release.

When an app is executed, Android initiates a Linux process for it. The main thread within this process runs a message queue using Handlers to manage top-level app components, e.g., Activities and BroadcastReceivers [3, 20]. Android allows apps to offload blocking or long-running tasks to the background (with some restrictions), or to schedule them based on specific criteria, using specific APIs, e.g., services, WorkManager, and AlarmManager. Below we provide

[3] As of Feb 2026, according to one estimate, Android 13 is the third most popular Android version, with a 14.5% market share; see https://www.appbrain.com/stats/top-android-sdk-versions.

an overview of these APIs and restrictions as pertinent to our methodology and implementation.

Services. Services in Android are designed for long-running operations and can operate in various modes. *(1) Foreground services* manage tasks that require the user's awareness of continuous execution, where uninterrupted operation is crucial for the user experience (e.g., navigation apps). These services must be highlighted by a persistent notification in the status bar within 10 seconds of their start. If the execution duration of a foreground service is less than 10 seconds, showing a notification is not necessary, and the system considers it a short-running foreground service [23]. *(2) Background services* run without direct user interaction and are suited for tasks that can tolerate interruptions (e.g., periodic data syncing). They do not need a persistent notification, have lower priority in resource allocation, and are more susceptible to termination under resource constraints.

Background tasks. Android offers several APIs to manage background tasks, i.e., operations an app perform outside its main thread. *(1) Asynchronous operations* enable apps to perform concurrent tasks in the background without blocking the main thread (with different implementations, including coroutines and threads). Such tasks can be executed on individual threads, or utilize executors [13] to manage their execution within the same thread, or through thread pooling systems to improve efficiency. As its lifecycle is bound to the initialized component [6], asynchronous work can be executed only when the app is in the foreground. *(2) Task scheduling APIs* manage tasks that can be deferred/interrupted during execution, based on the device status even if the app is in the foreground [17]. These tasks can run for extended periods (e.g., 10+ minutes), and managed by *WorkManager* and *JobScheduler* APIs [22]. WorkManager is designed for deferrable, guaranteed execution (e.g., sending logs to a server, even after a device restart), utilizing JobScheduler [24]. JobScheduler optimizes resource usage by scheduling jobs based on specific criteria (e.g., uploading data only when the device is charging and connected to Wi-Fi), without guaranteed task completion, as tasks may be lost after a restart [16]. *(3) Broadcast receivers* utilizes intents, allowing an app to listen for messages from the system, other apps, or the app itself based on system/app states, and user actions. For instance, an app can use a BroadcastReceiver to initiate a background work (with a duration of 10 seconds by default), when the device's network connectivity changes or when the device boots up. Broadcasts are categorized by their registration mechanism, dynamic or static, which dictates their operational lifecycle. Dynamic receivers are registered at runtime via the registerReceiver() API and only process intents while the application process is actively running. Intents themselves are classified as either explicit, targeting a specific component, or implicit, broadcasting an action globally to any matching receiver. Conversely, static receivers are declared within the AndroidManifest.xml, allowing the system to instantiate the application to handle specific intents even when it is not currently executing. However, to mitigate background execution overhead, Android 8.0 introduced stringent resource limitations: static receivers are now restricted to a whitelist of exempted implicit broadcasts, forcing applications to rely on dynamic registration for most system-wide events [11, 14, 54].

Alarms. Alarms in Android, managed via the AlarmManager API, are mechanisms that trigger tasks at specific times or intervals and can wake the device from sleep to execute time-sensitive operations. Alarms themselves do not execute any process if the app is not running; instead, they send an intent to the defined component, such as a BroadcastReceiver, to execute the work [1].

Restrictions on background works. *(1) Doze mode* conserves battery by minimizing background activity when the device is stationary, not charging, and the screen is off. During doze mode, background works are deferred, and network access is suspended, with brief maintenance windows allowing for necessary updates. *(2) App standby buckets* classify apps based on usage patterns to manage background activities and battery consumption. The classification ranges include: active apps (no restrictions), working set apps (slight restrictions), frequent apps (more significant restrictions), and rare apps (the most stringent limitations, such as delayed notifications and limited background activity [5]). *(3) App background restrictions*, starting with Android version 8.0, major limitations were introduced to prevent apps from starting background services when not in the foreground. Developers must use JobScheduler, WorkManager, or foreground services for background tasks. A newly created background service must invoke `startForeground()` within 5 seconds of its creation and provide a notification to the user of the transition to a foreground service. Location access is also further restricted for apps in the background (e.g., allowing locations updates only a few times per hour [7, 8]). Although Android continues to impose further restrictions in later versions [9, 10, 18, 19]. *(4) Battery optimization* extends battery life by placing apps in different optimization states based on their activity and resource usage. Apps that consume excessive battery in the background are flagged, prompting user action, or system restrictions (e.g., allowing only essential background activities).

2 High-Level System Design

In this section, we present a high-level overview of Panoptes, outlining its design and core functionalities. Panoptes simultaneously analyzes multiple apps on a single device, identifying privacy exposures in background works and the specific events that trigger them. It operates in two primary phases: data collection and data processing. Before initiating the analysis, the environment is prepared by downloading and configuring the target apps, setting up the test devices, and deploying the necessary modules or techniques to bypass potential evasion mechanisms; see Fig. 1 for an overview (for the execution flow see Fig. 2, detailed in Sec. 3).

Data collection. This phase consists of two key components: the *gateway* and the *agent*, which are designed to execute target apps and trigger background work while monitoring app activities. The gateway, deployed on a computer, oversees the entire data collection process, while the agent, running on an Android device, communicates with the gateway to receive commands and relay logs. To manage extensive logging requirements, the gateway and the agent communicate via the WebSocket protocol.

The *gateway* serves as the central control unit, initiating both a WebSocket server and a proxy server to interface with the agent. It efficiently manages the simultaneous analysis of multiple apps on the same device. The gateway begins by scanning the apps' manifest files to identify relevant services and BroadcastReceivers.

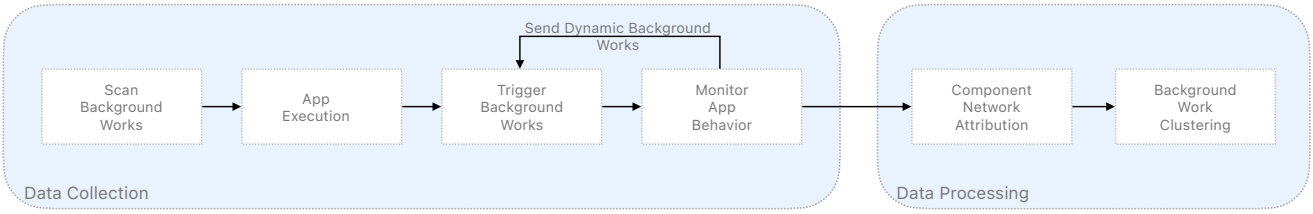

Figure 1: Panoptes overview

Following the installation of the apps and the granting of necessary permissions, the gateway sequentially starts the services and triggers BroadcastReceiver events. It then listens for logs from the agent, and activates jobs and other background works registered during app execution, including those registered dynamically.

The *agent* operates on the test Android device, capturing app activities at both kernel and user space (Android Runtime) levels by utilizing the eBPF and LSPosed frameworks. It instruments kernel level operations by leveraging kprobes, tracepoints, and cgroups program types. These operations encompass packet forwarding, network metadata collection, and process management activities, including forking, executing, and exiting processes. We attribute each action to its corresponding threads and processes across various apps. Complementing the kernel level instrumentation, our user space instrumentation provides extensive logging capabilities. By hooking into hundreds of Android SDK APIs, we achieve the ability to trace asynchronous operations, monitor background work APIs, capture encryption and decryption activities, and simulate GPS coordinates. Additionally, the user-space module is designed to capture scheduled background tasks upon registration and transmit them to the gateway. This enables the gateway to place these tasks into a pool for subsequent triggering, thereby effectively tracing asynchronous operations and background processes.

Data processing. Once the dynamic analysis is completed and the network traffic along with the logged app activities obtained, Panoptes separates the network connections of each app based on records obtained from API hooking. Then it performs deep packet inspection to identify any data exposures and assess the status of the sockets during execution, determining whether connections originated while the app was visible or running in the background. Panoptes further examines whether the app utilized background works, attributing these connections to specific app behaviors. Finally, it compiles statistics on prevalent scenarios in which apps initiate connections and transmit data in the background, uncovering privacy exposures (if any).

3 Implementation

In this section, we detail the implementation of Panoptes. We first discuss the setup steps for our test environment, and provide a brief overview of Panoptes's core phases. We then explain in detail how we systematically trigger/monitor various types of background works, and asynchronous processes. Finally, we also discuss various challenges and our solutions for effective instrumentation.

Panoptes is implemented using Rust, Kotlin, and Python, totaling ≈14,607 lines of code. The choice of languages is driven by the technical constraints of each component. We use Kotlin (LOC: 3334) for the LSPosed module as it requires integration with the Android JVM. For kernel-level monitoring via eBPF, we use Rust (LOC: 1078), as it provides the systems-level control required for kernel probes. Rust is also used for the high-performance dynamic analysis (LOC: 3493) and packet inspection (LOC: 6286) modules. Finally, we use Python (LOC: 416) for manifest extraction and preprocessing, where library support and development flexibility are prioritized over execution speed.

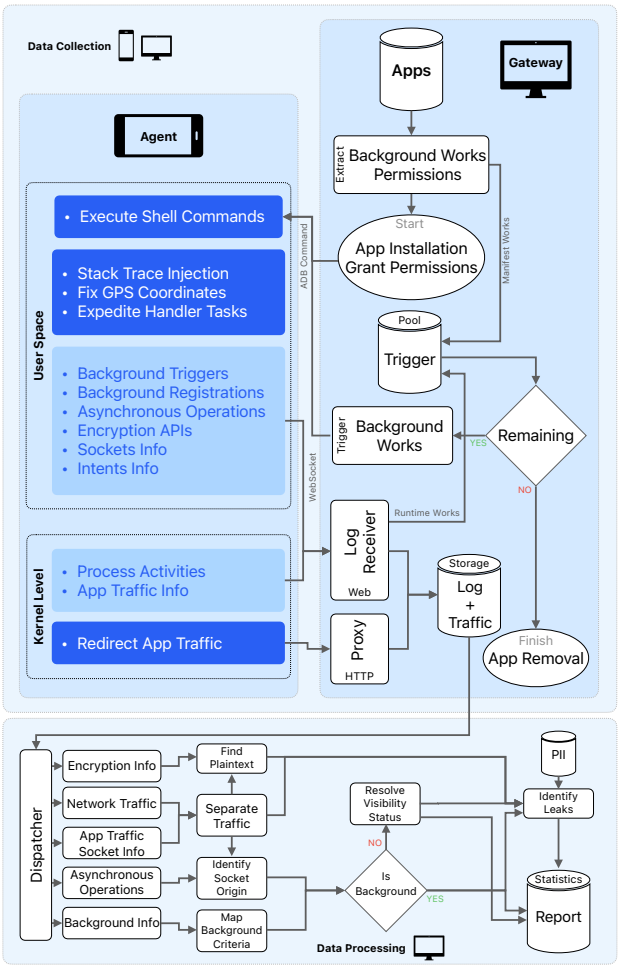

Figure 2: Panoptes execution flow

3.1 Setup and Implementation Overview

Test environment setup. Panoptes requires root access on the test devices to perform API hooking via LSPosed, kernel-level instrumentation using eBPF, and to send system-level intents. To obtain root access, we patch the device's stock framework using Magisk [35], followed by installing the LSPosed framework user space instrument. We install our mitmproxy CA certificate (for capturing plaintext HTTPS traffic), and use the BypassRootCheck-Pro [44], and JustTrustMe [42] modules within LSPosed to conceal root access, and accept invalid certificates from our proxy; we verified the effectiveness of root detection evasion using RootBeer [48]. To handle situations where some apps may disable Wi-Fi or switch to airplane mode, we create a Magisk service to execute `svc wifi enable` and `settings put global airplane_mode_on 0` every second to ensure the persistence of the connection. Moreover, we noticed that Android sometimes automatically disconnects Wi-Fi when captive portal detection fails. To disable this check, we run `settings put global captive_portal_mode 0`. Finally, we remove the device's lock screen and change the display timeout to 30 minutes to prevent battery optimization effects and background restrictions during our analysis. The agent also hooks the `getLatitude` and `getLongitude` methods in the Location class to always return fixed values, enabling us to maintain consistent coordinates during data processing.
Data collection pipeline. First, a list of target APKs are downloaded using the Apkeep tool [30]. We then use a Python script to organize these APKs, ensuring they follow a consistent format as they are downloaded from different sources. This script also utilizes jadx [53] to extract the AndroidManifest.xml files from the APKs. Then our gateway scans the extracted manifest files to identify permissions, services, and BroadcastReceivers. The apps are then installed sequentially, with up to 8 test apps installed concurrently on each test device (using the `adb install-multiple` command). We automatically grant all the permissions required by the apps.[4]

The gateway then launches a proxy server using mitmproxy [39] and runs each app for 10 seconds on foreground screen without interaction before minimizing it, to monitor the background works registered during app execution. Note that the default Android timeout for app loading is 5 seconds, and we chose 10 seconds as grace period to ensure that each app is fully loaded before minimizing it. Note that, we do not perform any UI interaction, and as such explicit privacy dialogs (e.g., for third-party data sharing) are neither accepted nor dismissed; the captured network traffic thus reflects the app's default behaviour. To analyze apps that use *custom encryption* (e.g., ad-hoc encryption of content transmitted over HTTP or HTTPS), we relied on the method introduced by ThirdEye [49]. We implement a Java API cryptographic operation logger using LSPosed. The logger hooks apps' cryptographic APIs to intercept plaintext, ciphertext, and associated cryptographic parameters at runtime. By correlating these plaintext-ciphertext pairs, Panoptes identifies and isolates specific encrypted segments within the captured network traffic. Panoptes then logs device behavior, including background work activities, and proceeds to install the next app while the previous ones are minimized and under analysis. It then triggers the registered background works identified from the AndroidManifest.xml file, and monitors dynamically registered background works to trigger them subsequently. This process continues until all identified background works have been triggered.
Data processing pipeline. Once dynamic analysis is completed, we process the network traffic captured by mitmproxy. We reconstruct values obtained from the custom encryption APIs that we hooked, followed by performing network packet inspections to identify PII transmission. We also attempt to decode and decompress the data obtained from the network using various algorithms, including hex, base64, zip, gzip, zstd, based on their detected magic numbers. When network inspection is finished, we attribute the origins of these connections to three categories: the app was visible, the app was not visible, or the connection was established by background works. For the last case, we perform further attribution of the network traffic to identify which background work, which class, and what criteria triggered such events.

[4]Using commands, e.g., `pm grant`, `appops set`, `cmd notification`, and `settings put`.

3.2 Triggering Background Works

Background works in Android apps are typically triggered based on criteria defined by the app developers. However, these criteria may not be satisfied during our analysis on their own (e.g., switching from 4G to 5G). Therefore, we investigated Android's background work mechanisms and developed methods to trigger background works prematurely (while the app is not in the foreground), via Services, JobScheduler, WorkManager, BroadcastReceivers, and AlarmManager. For an overview of these components, see Sec. 1. Note that the privacy exposures detected through these triggers genuinely reflect an app's behaviour—Panoptes simply expedites such behaviours for scalable measurement.
Services. Foreground and background services generally use identical parameters and attributes in the manifest file, except when foreground services require additional parameters. This similarity makes it unreliable to distinguish them solely based on the manifest. Therefore, we treat all services declared in the manifest file as targets to trigger, regardless of their actual type, and monitor the invocations of `startForeground` method to determine whether a service is intended to function as a foreground service; if this method is invoked, the service is classified as a foreground service. We execute services by issuing system-level commands via ADB commands. We also monitor the invocations of `onStartCommand` and `onBind` methods in all classes that implement the `Service` abstract class to log the process ID and execution time, mapping this information to network packets.
JobScheduler. JobScheduler allows the scheduling of asynchronous jobs that need to run in the background. Each job scheduled is assigned a unique ID (as integer). In Android, we can only trigger jobs using their jobid after they are set. Thus, we hook the `jobScheduler.schedule` method during the scheduling process to capture the job's execution criteria and its assigned ID. Additionally, we hook all the classes that implement the `JobService.onStartJob` interface to collect the job ID for mapping the job criteria and the process ID to correlate with network traffic. After setting the hooks, we can obtain the job ID during execution from the `schedule` method, which we use to trigger these scheduled jobs.
WorkManager. WorkManager, like JobScheduler, facilitates the scheduling of deferrable, asynchronous background works. Each

work scheduled via WorkManager is assigned a unique ID (as UUID). WorkManager does not provide a direct way to trigger works using their work IDs via ADB. We investigated the internal implementation of WorkManager and identified the method[5] responsible for converting works into jobs and assigning job IDs during the scheduling process. By hooking into this method, we could capture both the work ID and the job ID, and map them together during the conversion. This mapping enabled us to utilize ADB commands to execute these works in the same manner as JobScheduler jobs.

BroadcastReceivers. BroadcastReceivers listen for receiving either system-generated intents (e.g., battery low), or app-generated intents (e.g., logout from an SSO account). We first scan the AndroidManifest file to identify statically registered receivers. Then we hook all variants of `registerReceiver` function to receive any dynamically registered broadcasts. We then hook the `onReceive` method in all classes that implement `BroadcastReceiver` to capture the process ID and work information, mapping them to network connections. Prior to triggering these receivers, we attempt to resolve the intent value of the target action, if available. We then trigger the receivers with root access, as most system-related broadcasts cannot be generated with standard ADB permissions. We observed that sending broadcasts immediately places a high load on the OS, often causing device freezes or system restarts when multiple broadcasts are sent concurrently. To avoid this, we trigger such receivers asynchronously, deferring the broadcast until the system can process it. However, broadcasts must be executed within 10 seconds by default, or they are discarded by the system. To manage this, we explored Android's internal commands and found that using the `dumpsys activity broadcasts` command under the `Deferred Ordered` subgroup allows us to monitor pending broadcasts. Before sending a new broadcast, we check for existing pending broadcasts; if any are present, we wait until they are processed. Additionally, we utilize specific flags when sending broadcasts[6] to minimize performance issues and broadcast cancellation.

AlarmManager. The AlarmManager itself does not execute work directly; rather, it schedules the sending of an intent at specific times, allowing activities, services, or BroadcastReceivers to start, even when the app is not running. Unlike other background tasks, expediting `AlarmManager` jobs isn't needed since Panoptes already handles broadcasts and services, covering spontaneous alarm-based triggers as well. However, this approach does not enable us to definitively attribute specific work executions solely to an alarm trigger. To address this, we examined the Android source code and found that all alarm-setting methods eventually call the internal `setImpl` function in `AlarmManager`. Hooking this single function captures time data and `PendingIntent` IDs with minimal overhead. Since `PendingIntent` conceals its target intent, we also hook its service and broadcast creator functions to map IDs and identify the corresponding background work.

Handling Android Intents. We cover BroadcastReceivers and services, both of which use intents for message handling. We identify the actions of these intents from the manifest file or during their run-time registration. However, some intents may require additional data; for instance, the `NEW_OUTGOING_CALL` action requires a phone number provided in the `incoming_number` variable. Without this data, the target BroadcastReceiver in the app may not execute correctly and may even crash.

To obtain the necessary values for such intents, we monitor all intents received by BroadcastReceivers and services, alongside the intent values from all `sendBroadcast` methods from the Context class, and all service and broadcast methods from the `PendingIntent` class during analysis of the target apps. When an intent is received from these objects, we store its values in a database (without duplicates). For future actions sent by Panoptes, we first check the database for these values; if they exist, we trigger the broadcast with the retrieved values, based on their registration within the target app.

To gather baseline intent values prior to our main analysis, we first developed a test app that registers all documented Android intents and protected broadcast intents extracted from the Android source code, yielding 864 unique intents. We ran this app on a device and manually interacted with various system menus (e.g., changing the timezone) to induce OS-level broadcasts. To further expand coverage, we then remove the test app and performed a preliminary analysis across all apps in our dataset to capture any app-generated or app-received intents. In total, we resolve 21,907 unique intents before proceeding with our main experiment.

Cross-background work execution. Background works can invoke one another, complicating our attribution process. For example, a BroadcastReceiver, upon receiving an event, can initiate a service; then it becomes difficult to attribute traffic from the service to the event received by the BroadcastReceiver. To address this, we collect stack traces of all the components that start and register these background works, allowing us to create an execution tree, and accurately identify the parent work and its caller.

3.3 Attributing Asynchronous Operations

Android inherently relies heavily on asynchronous operations, utilizing various techniques e.g., OS threads, Android Handlers [3], and Kotlin coroutines. We use thread and process IDs with stack traces to attribute the sockets created by an app to their corresponding background works, which presents several challenges. When any app component is executed, its main thread is responsible for running a message queue via a Handler; app components submit works to the Handler, which then executes them within its thread and process ID. However, after execution, the Handler process may not terminate, and wait for new messages; e.g., the Handler may process a broadcast message, and then continue to receive and handle other broadcasts until it is terminated, resulting in multiple messages being processed under the same thread and process ID.

Network connections also often utilize asynchronous operations, further complicating stack trace analysis. Asynchronous network requests are executed within the context of Handlers or other task executors, not within the same execution context as the app component that initiated them. This behavior makes it challenging to trace the origin of these requests because their stack traces contain references to the Handlers/executors rather than the originating

[5] androidx.work.impl.background.systemjob.SystemJobScheduler.scheduleInternal

[6] Such as: `FLAG_RECEIVER_NO_ABORT` (preventing the broadcast from being aborted if it is an ordered broadcast); `FLAG_RECEIVER_INCLUDE_BACKGROUND` (allowing the intent to be sent to a background work); and `FLAG_ACTIVITY_MULTIPLE_TASK` (starting the work even if it has already been started).

code. Apps also employ various mechanisms to manage asynchronous operations, e.g., thread pools or executors, which can recycle threads and execute multiple works within the same thread and process ID, further complicating traceability.

To map network connections, we initially adopted the strategy of executing every asynchronous operation in a newly spawned thread (similar to Jianfeng et al. [37]). This migration, however, destabilized many apps and repeatedly raised a runtime exception–"Can't create handler inside thread that has not called Looper.prepare()." Because the relocated tasks no longer share the main thread's `Looper`, the method proves unsuitable for large-scale dynamic analysis.

We therefore introduce an approach to trace connections initiated from both Java and native code. We first embed distinctive metadata into the task's stack trace during task submission to executor, enabling us to link every network request back to its origin to trace Java code. Then we extend the idea of Jianfeng et al. [37] by *cloning*—rather than migrating—the background task at runtime and launching the clone in a separate thread with a unique identifier. The cloned path exposes native-level operations while the original task proceeds unhindered. Implementation details follow.

Stack trace injection. When an asynchronous work is executed, its stack trace does not contain information about the component that created the work; instead, it reflects the executor's stack trace since the code runs within its context. For example, a BroadcastReceiver might create a network request after receiving an event, but since the network request uses asynchronous operations, it is executed by the Android Handler. As a result, the stack trace for the socket shows a Runnable class used in the Handler, not the BroadcastReceiver, as the origin (see Listing 1 in the appendix).

To address this issue, we first attempted to modify asynchronous operations during their submission to the executor, adding a record to its stack trace to map it back to the originating code. However, Android restricts direct modification of stack traces. Therefore, we explored the asynchronous Android APIs and found that the majority of asynchronous operations rely on Java's Runnable and Callable interfaces. We then created wrapper classes for these interfaces. Instead of submitting the original object directly to the executor, we submit a wrapped object that invokes the original object internally. Using this wrapper, we could inject additional information into the stack trace. However, using a single wrapper for all scenarios is insufficient because a constant record in the stack trace does not provide enough information for precise attribution. Therefore, we dynamically generate unique classes for each submission. The class names include the process ID and thread ID. We also examine the submitter's stack to determine if it originated from a background work component. If so, we injected an identifier indicating the type of background work along with a random value as part of the class name. We also collected the corresponding stack trace alongside these injected identifiers. This approach allows us to map asynchronous works back to the app component that created them, as the stack trace now includes our injected record, enabling precise identification of the work's origin.

However, creating a new class for each asynchronous operation submission via API hooking poses significant challenges. We initially attempted to use Java libraries like Javassist [34] and Java ASM [56] to dynamically create classes, but these approaches failed because the generated code could not be loaded into the app. Consequently, we opted to create wrapper Java classes for the Callable and Runnable interfaces off-device, compiling them into DEX files (with identifiable class names) that could be loaded into the app at runtime. The class names included placeholders for the process ID, thread ID, an identifier (to distinguish background work types), and a randomly generated marker value. These placeholders are populated with actual values during execution. The precompiled classes are hard-coded into the agent. When a record injection is needed, the appropriate DEX file is retrieved, and the placeholders are populated with the actual values before the class is loaded into the target app and used as a wrapper.

The final class name in the stack trace appears in the format of StackTracer_1hjXH6f6_13243_13556 for operations originating from background work, or StackTracer_____13243_13556 for others. To optimize performance, we check for the existence of the dynamically generated classes in the system and only generate them if they do not already exist. Listing 2 (in the appendix) shows the stack trace of a socket that was initiated by a BroadcastReceiver and executed as a new thread.

Note that the integrity of the modified DEX files must be maintained, as Android verifies DEX file correctness using Adler-32 and SHA-1 checksums. Modifying the DEX file during execution could invalidate these checksums. To maintain integrity, we developed mechanisms to recalculate these values after modifying the placeholders (following the DEX file structure and Android documentation [12]).

Preserving original class names in hooked method stack traces. We observed that when methods are hooked, the original method is invoked internally, but the stack trace omits the actual class names of the calling classes; see Listing 3 (in the appendix) for the stack trace from the `onReceive` method of a BroadcastReceiver—the stack trace shows a generic hook class (`LSPHooker_`), instead of the actual class `com.app.view.receiver.WakeReceiver`. We thus modified LSPlant, the Android ART hook library utilized by LSPosed, to retain the original class name during hook creation. We enhanced this function to append the original class name to the default hook name by leveraging JNI interfaces during the hook creation; see Listing 4 (in the appendix) for an example.

Executing background works in a new thread. The stack trace injection effectively works on APIs whose components are invoked in the Java part of the app. However, a background work might utilize the NDK to establish a socket, in which case the Java stack trace of the socket cannot be obtained. Moreover, background works primarily execute on the Handler thread and share the same thread and process IDs with other components, making it challenging to attribute them to their parent works. To cover native sockets initiated by background works, during the registration of background works, we duplicate each background work and assign it to a new thread for execution while allowing the original instance to continue its operation. Executing the additional work in a new thread enables it to have a unique thread ID instead of executing within the context of the Handler or executor, which enables us to map the connection to its background work based on their thread ID. However, we allow the actual background work with its Handler or executor to execute, as executing them in different threads might introduce crashes. Additionally, we implement exception handling

for newly created threads to prevent app crashes if thread creation fails. Although this approach is not perfect, it improves our coverage, and identifies sockets in native code by comparing the thread and process IDs obtained from the kernel level instrumentation with those from the executed background works.

4 Results

In this section, we present our findings on the privacy risks associated with background works in Android apps. For our dataset, we leveraged AndroidRank, a platform that maintains the most popular apps across 33 distinct categories, ranked by download metrics. This list provides a relatively diverse sample of Android apps, in contrast to focusing solely on the most popular apps.

App statistics and analysis performance. We scraped apps in all 33 categories on Oct. 15, 2024 and collected 15,456 unique apps. The Apkeep tool was used to download apps from Google Play, and APKPure [25] served as an alternative when apps could not be downloaded from Google Play due to device compatibility or geographic restrictions. In total, we successfully downloaded 15,188/15,456 (98.3%) apps, with 9972/15,188 (65.7%) from Google Play and 5216/15,188 (34.3%) from APKPure. We successfully analyzed 14,044/15,188 (92.5%) of these apps. However, 1144/15,188 (7.5%) apps could not be installed; 1048 were incompatible with our test devices' CPU, while the rest (96) failed due to missing components or invalid signatures. All such failures occurred with apps downloaded from APKPure.

We conducted an initial run of Panoptes to collect intent information as a baseline, followed by the final run for privacy measurement. The final experiment was performed between Oct. 19–24, 2024, using two Pixel 7 Android phones running factory images with the latest version of Android 13. The analysis time per app ranged from 1 second to 117 minutes. Analysis of 11 apps finished in less than 1.5 seconds; these apps crashed immediately and did not contain background work components in their AndroidManifest.xml file. The longest analysis times were for apps that extensively utilized background works, both defined in the manifest file and registered dynamically. The median and mode of the execution times were 275 to 281 seconds, respectively, indicating that most apps required between 4-5 minutes; see Fig. 3 (in the appendix). We could analyze up to eight apps on each device concurrently, which significantly reduced our analysis time: about 5 days to test all apps in our setup vs. 45 days (65,689 minutes) without our optimization (sequential app execution). In total, we collected 39GB of behavior logs and 33GB of network data, with the dataset compressed.

4.1 Characteristics of Background Works

Although we triggered the execution of numerous background works, we primarily report statistics on background works that attempt to establish network connections during our analysis. Specifically, 13,884/14,044 (98.9%) of the analyzed apps attempted to establish at least one network connection, regardless of their visibility status or whether the connection was successfully established or dropped. We found that 13,068/13,884 (94.1%) of the analyzed apps established network connections while being out-of-sight (i.e., minimized or using background works). Among these, 13,038/13,068 (99.8%) established connections when the app was minimized, and 7534/13,068 (57.7%) did so via background works. Notably, 13,068/14,044 (93.1%) apps established network connections either when minimized (without relying on background works) and through background APIs, regardless of visibility.

Prevalence of background work components. To understand the criteria that apps use to trigger their background work execution (from a closed-state), and initiate outside connections, we rely exclusively on statistics obtained from background work components.[7] We found that 411/7534 (5.5%) utilized services, 7144/7534 (94.8%) utilized JobScheduler or WorkManager, and 861/7534 (11.4%) utilized broadcasts; see Table 1. Notably, 3898/7144 (54.6%) apps exclusively established their connections using JobScheduler only when they were visible (within the 10-second grace period). The number of jobs in the visible status is higher because Panoptes triggers them as soon as receiving their registration event, and these apps schedule their works immediately upon opening (see Table 2 for their triggering criteria). In contrast, minimized-state apps schedule their works to run at a later time, rather than immediately after launching. We observed the registration of background work using AlarmManager in 108 apps; however, we did not observe any of them establishing network connections. Below we provide detailed results on different types of background works.

App Status	Connection	Services		JobScheduler	Broadcasts		Any
		Foreground	Background	WorkManager	Static	Dynamic	
(V)isible	(E)stablished	8	123	6301	9	246	6451
	(D)ropped	3	34	361	6	48	429
	Common ($E \cap D$)	1	11	291	2	21	340
	Exclusive ($V \Delta M$)	4	65	3898	5	189	4057
	Total ($E \cup D$)	10	146	6371	12	273	6540
(M)inimized	(E)stablished	43	269	3092	245	486	3647
	(D)ropped	14	78	373	69	135	578
	Common ($E \cap D$)	5	55	219	44	77	387
	Exclusive ($M \Delta V$)	46	211	773	263	460	1509
	Total ($E \cup D$)	52	292	3246	270	544	3838
Any	(E)stablished	46	322	7054	249	665	7422
	(D)ropped	17	98	599	72	170	835
	Common ($E \cap D$)	6	62	431	44	94	723
	Total ($E \cup D$)	56	357	7144	276	733	**7534**

Table 1: Prevalence of background work components in network connections.

Services: long-running operations. For the apps using services, we found that 56/411 (13.6%) of them invoked `startForeground()` to promote their services to foreground services. 13/411 (3.2%) of the services registered *dynamic broadcasts* to listen for events received from the system. Note that background services are typically unable to operate when their associated app is not running (discussed in Sec. 1). However, if triggered by other background works, these services can continue executing for up to 5 seconds without transitioning to a foreground state. Moreover, if the app is active but invisible to the user (e.g., such as when minimized) these services can persist and continue their operations. This mechanism enables services to maintain long-running tasks, including continuous network communication, without requiring user interaction.

Jobs: time and status driven operations. Our results clearly show that the use of background works via `JobScheduler` and `WorkManager` dominates other background work mechanisms, possibly due to the high flexibility they offer in terms of device status and timing

[7] Note that minimized-state exposure that does not utilize background work components generally lack defined triggers.

constraints. We successfully attributed jobs in 5840/7144 (81.7%) apps and identified their execution criteria (see Sec. 5 under "Limitations" for a discussion on unattributed jobs). Jobs are inherently time-driven; by default, they run immediately after scheduling if no timing parameters are set. However, jobs can be configured to run once with a specified latency (delay), or to run periodically.

We found that works scheduled directly with `JobScheduler` tend to execute immediately, as they are designed to do so by default. Interestingly, we observed that the majority of jobs originating from `WorkManager`—which internally transforms work requests into jobs—execute with an initial delay not documented in the official Android documentation. To understand this behavior, we investigated the `WorkManager` component in the Android source code. We found that during the transformation of a work request into a job, `WorkManager` internally calculates a minimum latency based on the initial back-off time, which is the waiting period before retrying work execution. Consequently, jobs initiated from `WorkManager` execute with a minimum delay of 10 seconds, with a default value of 30 seconds.[8] Due to this behavior, we observed that many apps execute their jobs with this delay. Additionally, for the apps where we could attribute the jobs, we found that 5690/5840 (97.4%) apps check the device's network status before executing their jobs and establishing connections. Furthermore, we observed that 200/5840 (3.4%) apps set the `important` flag on their jobs to prevent background restrictions while they are running, despite its deprecation since Android 12 [21]; all these jobs were scheduled to execute immediately. Table 2 summarizes the scheduling constraints/delays we observed.

Constraints	One-time				Periodic				Any
	$\leqslant 10s$	$10s \leqslant 1m$	$1m \geqslant 30m$	$30m \geqslant$	$\leqslant 10s$	$10s \leqslant 1m$	$1m \geqslant 30m$	$30m \geqslant$	
Network	547	5004	13	112			1	49	5690
Battery	4			6				9	19
Charging	5		1	2				1	7
Device Idle				88					88
Storage				1					1
Expedited	1								1
Important	200								200
Overall	678	5005	15	118			5	55	5840

Table 2: Distribution of jobs based on scheduling constraints and initial delays (empty cells represent zero values). For instance, 88 apps (highlighted) execute their works only once after a delay of 30+ minutes when the device is idle; and 49 apps (highlighted) execute their works every 30+ minutes when certain network criteria are met. Results are not mutually exclusive: a single app may appear in multiple columns when it establishes a connection using different constraints.

BroadcastReceiver: event driven operations. For the apps that established network connections via background works, we observed that 861/7534 (11.4%) of the apps did so after receiving a broadcast. 743/861 (86.3%) of these apps utilized *dynamic receivers*, which require the app's process to be active for delivery. In contrast, 253/861 (29.4%) of the apps utilized *static receivers* to allow messages to be delivered even if the app is not opened. Note that some apps utilized both broadcast types. Notably, we found that 29 apps established connections via widget-related broadcasts. Although we did not enable any widgets during our analysis, we identified their definitions in the manifest files and were able to trigger these broadcasts to observe their behavior. These serve as static entry points because the OS wakes the parent app process to execute the provider's code, even if the app was previously inactive or closed. Additionally, we identified several undocumented implicit excepted broadcast during our experiments (see Sec. 5 under "Undocumented implicit broadcast exceptions").

Among the apps using dynamic broadcasts, 311/743 (41.9%) of them utilized Android system intents, indicating that apps (and their servers) are apparently keen on collecting information about the device's status and system events. While dynamic broadcasts are typically considered catalysts rather than entry points, as they usually require an existing active process, they remain effective for apps running in a minimized state. Our results show that among dynamic broadcasts, connectivity-related events are more prevalent, whereas checking the device boot status is more common in static broadcasts; see Table 3.

Type	Action	Apps
Dynamic Broadcasts	android.net.conn.CONNECTIVITY_CHANGE	237
	com.android.vending.INSTALL_REFERRER	100
	com.batch.android.runtime.session.new	38
	android.app.action.NOTIFICATION_CHANNEL_BLOCK_STATE_CHANGED	34
	android.app.action.NOTIFICATION_CHANNEL_GROUP_BLOCK_STATE_CHANGED	28
	androidx.work.diagnostics.REQUEST_DIAGNOSTICS	28
	android.intent.action.QUICKBOOT_POWERON	23
	android.intent.action.USER_PRESENT	22
	android.intent.action.SCREEN_OFF	21
	android.intent.action.PACKAGE_ADDED	17
	android.intent.action.PACKAGE_REPLACED	13
Static Broadcasts	android.intent.action.BOOT_COMPLETED	135
	android.intent.action.MY_PACKAGE_REPLACED	65
	android.app.action.APP_BLOCK_STATE_CHANGED	32
	android.intent.action.TIMEZONE_CHANGED	31
	android.intent.action.LOCALE_CHANGED	31
	android.appwidget.action.APPWIDGET_UPDATE	29
	android.intent.action.TIME_SET	27
	android.intent.action.MEDIA_MOUNTED	6
	android.intent.action.PHONE_STATE	6
	android.intent.action.NEW_OUTGOING_CALL	6
	android.intent.action.MEDIA_REMOVED	5

Table 3: Top broadcast actions captured by apps via dynamic and static receivers. The highlighted row indicates undocumented implicit broadcast exceptions.

For apps using static broadcasts, `BOOT_COMPLETED` was the most prevalent, used by 135/253 (53.4%) apps. This event is broadcasted when the device completes its boot process. Based on the nature of autostart mechanisms and internet connectivity, we hypothesize that apps use this trigger to notify servers to send telemetry data and register the device with the Firebase Cloud Messaging (FCM) for push notifications. For dynamic broadcasts, `CONNECTIVITY_CHANGE` is sent when network connectivity changes, such as switching from cellular data to Wi-Fi. In this case, we expected to observe apps starting their data synchronization processes. However, we found that both dynamic and static broadcasts are often misused to track users, with instances of sensitive personal information (PII), such as device identifiers, Wi-Fi BSSID, and GPS coordinates, being transmitted without restraint; see Sec. 5 for prominent examples.

Third-party recipients. Among the analyzed apps 12,637/14,044 (90.0%) successfully established network connections to external endpoints (app servers or third-parties). Of these apps, 11,689/12,637 (92.5%) made such connections when they were visible, 10,502/12,637 (83.1%) when minimized, 7461/12,637 (59.0%) when using background works, and 11,377/12,637 (90.0%) when either the apps were

[8] For latency calculation, see the SystemJobInfoConverter class in the Android source code.

minimized or utilizing background works (out of sight). Please note that there is some overlap among these categories.

To understand the prevalence of third-party domains, we set a threshold: if a domain (including its subdomains) appeared in 10 or more apps, we label it as a potential third-party domain.[9] We found that 12,287/12,637 (97.2%) of the apps connected to third-party domains. Specifically, 11,480/11,689 (98.2%) did so when the app was visible, 10,028/10,502 (95.5%) when the app was minimized, 7155/7461 (95.9%) when using background works, and 11,075/11,377 (97.3%) when either the app was minimized or using background works; Table 8 (in the appendix) lists the top 20 third-party domains contacted by the apps. In contrast, we observed 1301 apps connected to hosts that are first-parties while utilizing background works.

We utilized an open-source community-managed domain list for companies [45] to map the third-party domains to companies. We found that Google and Facebook are the most frequent destinations, with 12,217/13,884 (88.0%) and 4372/13,884 (31.5%) connections, respectively. For background works exclusively, Google domains dominate with 6942/7534 (92.1%) connections. Among the Google domains utilized in background works, Firebase logging components such as `firebaselogging` and `firebaseinstallations` were the most accessed. These components are accessed through the app's own process ID and are not part of the Android platform or OS services. Besides the Google domains, we observed that many contacted domains are associated with tracking and advertising services, including those from Facebook (`graph.facebook.com`), AppLovin (`ms.applovin.com`), and Adjust (`app.adjust.com`).

Interestingly, we identified 34 apps that connected to third-party domains for tracking purpose *only when using background works*, and not when the app was in the visible or minimized states without background execution. Domains including `ad.simpledesign.ltd`, accessed by 18 apps, and `ad.leap.app`, accessed by 16 apps.

Nested execution. As documented [11], Android allows registering a work component using another one, e.g., starting a service from a BroadcastReceiver. Therefore, we monitor this mechanism to determine whether apps utilize it to create a work component that establishes a socket to the internet. We found that in 72/7534 (1.0%) of the apps where background works established internet connections, they were initiated by other background tasks. Specifically, 38/411 (9.2%) of the apps started such tasks from services, 14/7144 (0.2%) from JobSchedulers, and 22/861 (2.6%) from BroadcastReceivers (see Table 7 in the appendix).

4.2 Private Information Transmission

From our deep packet inspection of the collected network traffic, here we summarize what data is transmitted and how it is transmitted. Among the apps that successfully established at least one network connection, 12,214/12,637 (96.7%) apps transmitted privacy-sensitive information, including user and device data.

We categorized the data transmission practices based on the app's execution context: when the app is *visible* (foreground), when the app is *invisible* (in the background but running), when the app uses *background tasks* (e.g., services, JobSchedulers, BroadcastReceiver, excluding static broadcasts), and the *overall* occurrences across all contexts. Table 4 summarizes the types of data transmitted in these contexts, categorized into several groups such as *Device Information*, *Network Information*, *Network Location*, *GPS*, and *User Assets* (following the PII categories in [49]). Additionally, the table indicates whether the leakage in each context is exclusive, meaning that sensitive data is transmitted only through background works.

	Data Type	App is Visible		App is Minimized		Background Works		Overall
		Exclusive	Overall	Exclusive	Overall	Exclusive	Overall	
Device	Device ID	1252	1991	406	1106	80	268	2524
	Advertising ID	3187	4961	641	2347	48	281	5716
	Bootloader	112	124	87	102	20	25	249
	Fingerprint	205	1256	998	3958	3738	6862	8366
	CPU Model	146	163	284	303	30	32	498
	Display ID	1876	11489	434	8570	107	7203	12261
	Device Name	846	4081	1051	4938	2426	6893	9490
	Device Resolution	200	251	58	109	7	19	338
	Device ABI	1584	2634	1059	2067	173	381	3970
	Device Model	1748	11525	424	8877	104	7235	12284
	Device Timezone	1293	1893	796	1374	77	171	2859
Network	Operator	1348	1629	316	589	26	65	2021
	Device WiFi IP	270	333	46	110	8	14	396
	Device WiFi IPv6	51	56	47	52	3	3	106
	Default Gateway IP	1	6	3	8	3	3	12
	Device Proxy Address	122	517	2521	3308	1423	1963	4916
Location	Router ESSID	75	85	19	30	5	8	112
	Router BSSID	56	63	17	26	3	5	86
	Nearby Router ESSID	34	35	6	8	1	2	43
	Nearby Router BSSID	35	36	6	8	1	2	44
	GPS (≤7m accuracy)	452	567	70	168	18	50	658
	GPS (78m accuracy)	451	571	75	178	22	55	683
	GPS (787m accuracy)	470	600	102	216	29	63	747
User Assets	List of Device Apps	21	25	39	43	3	3	72
	Phone Number	25	27	44	46	4	5	75
	Call History Number	0	0	0	0	1	1	1
	Contacts Name	17	17	35	35	0	0	52
	Contacts Number	2	2	17	17	0	0	19
	Device Email	184	196	385	399	20	24	634

Table 4: Sensitive information transmission by apps across different states: when the app is visible, minimized, and during background works. "Exclusive" columns: data transmitted exclusively in that particular state, without prior or subsequent transmission in other states. We exclude dynamic broadcasts, as they cannot trigger app execution when the app is closed. Results are not mutually exclusive: a single app may appear in multiple columns when it transmits in multiple items/states.

It is evident from Table 4 that apps transmit sensitive data across all execution states (detailed below), including when they are out-of-sight (i.e., exposures via minimized-state and background works). Notably, background works are also responsible for a significant portion 7238/12,214 (59.3%) of this transmission (which can continue even when apps are closed/inactive).

Foreground (visible). When apps are in the foreground (only for 10 seconds), we did not perform any interaction with the app (including privacy-related prompts). Yet, 11,563/12,214 (94.7%) of the apps transmitted private information to their own or third-party servers, while the app was visible and without utilizing background work mechanisms. A substantial number of these apps transmitted device information such as *Display ID* (11,489 apps), and *GPS Coordinates* (600 apps).

Minimized-state (invisible). Even when apps are not visible on the foreground screen (minimized) and are not utilizing background works, they continue to transmit sensitive data. We identified that

[9] We adapted the threshold value from [50], which suggested setting a threshold of 5 hosts. However, we increased the threshold to 10, as some vendors might have several apps that share the same domains.

9056/12,214 (74.1%) of the apps transmitted private information while they were not visible; e.g., 3,958 apps transmitted *Device Fingerprint*, 399 apps transmitted the *Device Email*, and 46 apps transmitted the *Phone Number*. Although it is unclear whether this data transmission is intentionally performed while the app is minimized, some apps apparently check their visibility before initiating data transmission (see Appendix B).

Background works (visible/invisible). Background work mechanisms such as services, JobSchedulers, and BroadcastReceivers can facilitate data transmission without user interaction. We classified such practices regardless of whether the app was visible or not, as these mechanisms can operate even when the app is completely closed. 7238/12,214 (59.3%) of the apps transmitted privacy-sensitive information by using background works, regardless of whether they were in the foreground or not. We also successfully attributed leakage to the corresponding background mechanisms in 5831/7238 (80.6%) of the apps; see Table 6 in the appendix. The unattributed cases belong to JobScheduler (see Sec. 5, under "Limitations").

5 Discussion

In this section, we discuss prominent examples and the effects of device usage on out-of-sight privacy exposure. We also summarize several undocumented broadcast exceptions and the ineffective enforcement of app hibernation, which provide more opportunities for inactive or closed apps to execute their code. Finally, we discuss whether apps check their invisibility status and changes introduced in newer Android versions, followed by a list of our limitations. Other considerations, including app invisibility awareness, app hibernation enforcement, and the effect of user interactions, are provided in Appendix B.

Prominent examples. We identified several high-profile apps that leverage background works to collect and transmit sensitive user data while being out-of-sight. Some examples are as follows.

AccuWeather (package: com.accuweather.android, 100M+ downloads) was found to send GPS coordinates to `api.radar.io` upon receiving the `BOOT_COMPLETED` intent. That is, the app initiates background works to transmit location coordinates immediately after the device starts, without requiring the user to open or use the app.

Mi Telcel (package: com.speedymovil.wire, 100M+ downloads) utilizes two static broadcasts, `BOOT_COMPLETED`, and `SIM_STATE_CHANGED` to send detailed phone information to their endpoints, including: storage and memory capacity, screen details, device serial number, Android advertising ID, build ID, and device fingerprint. Note that `SIM_STATE_CHANGED` undocumented, and triggers on SIM card status changes, such as switching from/to airplane mode; see below under "Undocumented implicit broadcast exceptions".

Me - Caller ID & Spam Blocker (package: com.nfo.me.android, 5M+ downloads), advertised as displaying incoming call information and blocking spam calls, runs a foreground service[10] to transmit detailed call history information, including call durations, contact names and numbers, and call status (received/made).

Dating and Chat - SweetMeet (package: ru.fotostrana.sweetmeet, 50M+ downloads) uses JobSchedulers to transmit extensive device information every 24 hours when the battery has sufficient charge and the device is connected to the internet. It collects various data, including advertising ID, memory information, and display details.

RCTI+ Superapp (package: com.fta.rctitv, 10M+ downloads) uses JobSchedulers to transmit precise location coordinates every 4 hours when the battery has sufficient charge and the device is connected to the internet.

In addition to apps that utilize background work components as entry points when the app is not open, we also observed apps that rely on dynamic broadcasts requiring the app to be either visible or minimized. Below, we discuss two such apps that expose precise coordinates.

SHAREit: Transfer, Share Files (package: com.lenovo.anyshare.gps, 1B+ downloads) receives the `CONNECTIVITY_CHANGE` broadcast, which is triggered when the device's connectivity changes (e.g., switching from cellular to Wi-Fi), and subsequently sends precise GPS coordinates and device information (e.g., battery status, charging state) exclusively using BroadcastReceivers to `distribution.rqmob.com` over HTTP using custom encryption.

Salaat First: Prayer Times (package: org.hicham.salaat, 50M+ downloads), upon receiving the `CONNECTIVITY_CHANGE` broadcast intent—which is triggered when the device's network connectivity changes—transmits device information, precise GPS coordinates, and Wi-Fi network identifiers (BSSID and ESSID) to `hail-reporting.bronze.systems`. This transmission occurs without any user interaction, leveraging background execution to collect and send potentially sensitive location and network data.

Effects of device usage behaviors. We observed that the execution of background work components and the behavior of minimized apps are highly influenced by device usage behaviors (e.g., turning device on/off, setting an alarm). We found that apps can receive broadcast events and be executed upon booting (and unlocking) the device, even if it has not been opened by the user after installation—observed only in Android 13. For example, we observed that apps utilizing the `BOOT_COMPLETED` intent, can execute upon device boot without requiring any runtime permissions or user awareness. Specifically, we identified that 135 apps leverage this intent to establish connections to their endpoints, and 18 apps initiate the execution of their periodic or long-running works upon boot. Crucially, the timing of these events is directly influenced by device settings: for instance, while boot-related broadcasts fire immediately on devices without a lock screen, the OS delays these transmissions on protected devices until the user's first unlock. Beyond booting, environmental shifts act as persistent triggers. Transitions between cellular generations (e.g., 5G to 4G) or entering emergency mode trigger the `SERVICE_STATE` broadcast. Similarly, toggling airplane mode dispatches five different undocumented implicit broadcast exceptions, which apps may abuse as reliable entry points. Finally, 65 apps were found to leverage `MY_PACKAGE_REPLACED` to execute code repeatedly after updates—a trigger further compounded by the prevalence of enabled auto-updates.

Undocumented implicit broadcast exceptions. Since Android 8.0, to optimize device performance and protect user privacy, implicit broadcasts declared in an app's manifest file are generally not delivered to apps, except for a few cases as outlined in Android's documentation [14]. However, we found several undocumented

[10] "foreground service" refers to a service with a persistent notification and elevated priority; it can keep running even when the app isn't on screen

exceptions to this policy that allow certain implicit broadcasts to trigger background works, even when the app is not in the foreground or running. These broadcasts, listed in Table 5, cover a range of system events tied to device state changes, SIM modifications, and Google-specific service updates. Such events not only expose device status to apps but also serve as entry points for their code.

To identify the broadcast exceptions (static broadcasts), we collected all broadcasts and their corresponding actions defined in the app manifest files, checked documented intents, and examined the Android source code to gather system-protected intents from the `frameworks/base/core/res/AndroidManifest.xml` file. We also compiled all permissions listed in the app manifest files and created a test app using all collected permissions and intents. The test app included a BroadcastReceiver that logged executions for all listed intents without requiring dynamic broadcast registration. We installed this app on an Android 13 emulator, granted it all permissions, started it, and subsequently closed it; we then navigated through the device system menus while monitoring the app's log. Through this process, we observed that 13 of the received intent broadcasts were undocumented in Android's official exceptions. Further analysis across our dataset revealed that 3,772 apps utilized these undocumented broadcasts. Interestingly, we identified that `ACTION_MY_PACKAGE_REPLACED`, used in 3,441 apps, was not explicitly listed as exempted under certain background restriction contexts, yet we observed it was indeed received by our test app. The existence of these undocumented broadcasts introduces unintended entry points for app activity, especially since some broadcasts, such as `SIM_STATE_CHANGED`, are frequently triggered (e.g., during SIM card state changes or when toggling airplane mode) and do not require runtime permissions to receive.

Action	Event	Apps
android.intent.action.MY_PACKAGE_REPLACED	App update completed	3441
android.app.action.APP_BLOCK_STATE_CHANGED	App restriction change	454
android.intent.action.SIM_STATE_CHANGED	SIM status change	82
android.hardware.usb.action.USB_STATE	USB connection change	58
android.intent.action.SERVICE_STATE	Network state change	10
android.telephony.action.DEFAULT_SMS_SUBSCRIPTION_CHANGED	SMS SIM change	7
android.telephony.action.DEFAULT_SUBSCRIPTION_CHANGED	Default SIM change	7
com.google.android.checkin.CHECKIN_COMPLETE	Google check-in done	5
android.intent.action.ACTION_DEFAULT_DATA_SUBSCRIPTION_CHANGED	Data SIM change	5
android.intent.action.ACTION_DEFAULT_VOICE_SUBSCRIPTION_CHANGED	Voice SIM change	4
android.intent.action.SUBSCRIPTION_PHONE_STATE	Phone state change	4
com.google.gservices.intent.action.GSERVICES_CHANGED	Google services changed	2
android.intent.action.UID_REMOVED	App (UID) removal	0

Table 5: Undocumented implicit broadcast exceptions

Real-life background works execution. While we systematically trigger background works for scalable measurement, the actual magnitude of exposure in real life can deviate from our experimental baseline in two primary ways. First, our large-scale analysis was conducted without any user interaction or authentication. We expect a significant portion of apps to intensify personal data collection, in both diversity and frequency, once a user logs in and more sensitive information becomes accessible. Because many background tasks are registered dynamically only after an active session begins, these exposures remain largely unexplored in our study (for our limited experiment on the effects of user interactions without login, see Appendix B). Second, the overall frequency of leaks is moderated by individual habits and the user's unique behavior and device configuration. For example, while our baseline captures a snapshot of data exfiltration, the privacy risk in real life is a dynamic intersection of an app's logic and a user's expanding permission footprint. Granting specific permissions can drastically increase the activity of out-of-sight broadcasts; for instance, enabling access to SMS can enable apps to receive `android.intent.action.SERVICE_STATE` broadcast events, providing an additional, recurring entry point for data exfiltration that is not present in a clean installation.

Limitations. Our analysis lacks coverage of user interactions and certain app states (e.g., user login), which may leave certain privacy exposures or background activities dependent on specific states unexplored. Our traffic collection is limited to HTTP and HTTPS protocols, omitting other communication methods like raw TLS, TCP, and QUIC; consequently, we miss privacy exposure in these channels. Apps detecting root or dynamic binary instrumentation (DBI) may alter their behavior. While we verified our root-hiding via Rootbeer, advanced apps might still detect our LSPosed environment through custom logic. Consequently, the effectiveness of JustTrustMe and BypassRootCheckPro has not been tested. Our coverage of intent values is comprehensive but incomplete; some actions require context-specific values (e.g., for `ACTION_DEVICE_OWNER_CHANGED`) not consistently available in our setup, which may lead to gaps in execution and data transmission for some background works.

We also observed unexpected behavior in how WorkManager handles task execution, specifically regarding its unpredictable routing and opaque tracking mechanisms. For instance, in specific cases—particularly when works are triggered immediately, WorkManager bypasses the conversion of works to JobScheduler jobs. Moreover, as WorkManager operates as a library integrated within app code rather than as part of the Android framework, the workid-to-jobid mapping process can be obscured in apps employing rigorous obfuscation techniques. In such cases, while we were able to detect the occurrence of background tasks, we were unable to reliably trace their execution to the corresponding registrar functions or discern the specific criteria for their execution, particularly in the context of apps with highly obfuscated code structures.

6 Related Work

To the best of our knowledge, Panoptes is the first work to primarily focus on privacy exposure when an app is out-of-sight. Here, we summarize a few example studies from several comprehensive app-privacy measurement studies [29, 49, 51], and the ones that (at least partly) considered background works [26, 59].

Privacy-Sensitive data exposure detection. Early research established the ecosystem's risk profile by characterizing the pervasive misuse of sensitive identifiers and the deep integration of tracking libraries[31]. Building on this, Reardon et al. [51] analyzed 88,113 apps and identified various covert channels through which apps were transmitting sensitive information, such as GPS coordinates, device serial numbers, and IMEI numbers. They executed apps and monitored network traffic to identify inconsistencies between private data transmission and app permissions. The permission-based risks extends to investigations into VPN apps, which may facilitate unauthorized traffic interception or tracking due to the high-privilege nature of the associated system permissions [33]. Du et al [29] introduced MowChecker to address the issue of withdrawal inconsistencies, where users' choices to opt out of data sharing with

third-party libraries were often ignored. Their approach used taint analysis combined with manual verification to identify the withdrawal interface of apps and observe behavior changes when users altered this setting, detecting when apps continued to send data to third parties despite a user's withdrawal. Their evaluation revealed 157 such data collection inconsistencies in popular Android apps. These findings mirror broader investigations into systematic violations of GDPR's explicit consent requirements, where personal data is frequently transmitted to third-party entities prior to any user interaction or the provision of affirmative consent[40]. Finally, Pourali et al. [49] introduced ThirdEye, a dynamic analysis tool, particularly focusing on privacy exposures via custom encryption (any encryption other than protocol-based encryption like TLS) using a novel UI interaction method. Their analysis revealed that 2887/12,598 (22.9%) of apps transmitted sensitive information over the network using custom encryption.

While the aforementioned studies are comprehensive, they mainly address privacy exposures that occur during foreground operations (i.e., when the app is visible) and overlook background activities and the attribution of network connections to app components. As a result, a significant gap remains in the analysis of privacy exposures that occur in the background.

Privacy exposures in background works. Yang et al. [59] introduced AppIntent to identify whether a data exposure is associated with a user-initiated event or not. First, AppIntent utilizes static analysis to generate possible event sequences that could lead to data transmissions. Then, symbolic execution is employed to explore feasible execution paths, alongside generating data for intents, focusing on detecting whether the transmission was user-initiated or unintended. Finally, it automatically generates test cases for InstrumentationTestRunner [15] based on the data obtained from previous steps and repacks an app to execute the selected components. Then it runs dynamic analysis to execute tests and monitor the app's real-time behavior to verify whether the identified data transmissions occur as expected. Using AppIntent, the authors identified unintended data transmissions in 245/1750 (14.0%) apps, from a set of 1000 free apps and 750 malware apps. AppIntent covers only parts of background works, i.e., services and BroadcastReceivers, as the tool emphasizes GUI-driven, user-initiated events and lacks support for runtime events such as phone call triggers.

DroidJust [26] uses taint analysis to identify privacy exposures by linking sensitive data sources, such as user location or PII, to their respective sinks, where these data can be transmitted outside the application. Although DroidJust does not focus on detecting privacy exposures during background execution, it considers BroadcastReceivers as a potential source of sensitive data disclosures. In its evaluation, DroidJust identified 210/6111 (3.4%) applications exposing private data without clear component attribution.

While tools like DroidJust and AppIntent have contributed to detecting privacy exposures, they primarily focus on user-driven, foreground activities, leaving a critical gap in addressing background works in a comprehensive manner.

Android dynamic app tracing. To our knowledge, only Jianfeng et al. [37] address the problem of associating asynchronous network APIs with their originating components. The authors migrate every asynchronous task to a freshly spawned thread, enabling them to map each network request back to its caller. However, they did not report the success rate of this strategy, and our experiments show that it is impractical for large-scale dynamic analysis (see Sec. 3.3).

Beyond thread migration, existing dynamic data-flow tracers typically rely on either heavyweight static taint analysis or lightweight thread/process identifiers. A representative example is InviSeal [36], which is designed as a malware detection tool for emulated environments, emphasizing anti-emulation techniques and attributing network requests and other sensitive operations to specific app components through direct system call monitoring and framework-level API hooking. Schutte et al. [52] proposed AppCaulk as a data leak prevention mechanism, by repacking apps to inject targeted dynamic taint-tracking code directly into the Android app bytecode to trace sensitive data flows at runtime without requiring OS modifications. AppCaulk injects code for wrapper functions to enhance stack trace readability, focusing on data paths with potential exposure risks. However, the approach significantly increases app size (3–5 times), affecting AppCaulk's performance and app execution: out of 48 apps, only 15 were successfully installed and analyzed.

Summary of differences with existing work. We expand prior research by focusing on privacy exposure in Android apps when they operate out-of-sight, hidden from the user's attention, unlike earlier tools [26, 31, 33, 40, 49, 51], which mainly deal with user-visible foreground processes. Moreover, for tracing and attributing privacy risks, past work is either impractical [37] for large-scale analysis, fail to pinpoint the component responsible for it [36], or relied on static analysis techniques [52], including taint analysis and app repacking, which struggle to capture the complex, dynamic behaviors of background works. In contrast, Panoptes introduces a new real-time tracing mechanism designed to handle asynchronous processes, allowing it to pinpoint data flows without repacking apps. This dynamic approach enables more precise and immediate detection of privacy risks by observing asynchronous works directly. Overall, by carefully tracking data transmissions and expediting background executions based on their triggering criteria, we provide a more complete view of privacy exposures that occur out-of-sight.

7 Conclusion

We present Panoptes, a novel approach to identify out-of-sight privacy exposures in Android apps. Panoptes complements existing dynamic analysis tools (such as ThirdEye [49]) that focus on privacy exposure when an app is visible. When used in combination, the full picture of privacy issues in Android apps will become more apparent. By providing Panoptes as an open-source tool, we aim to facilitate both the research community and developers to investigate and reduce the identified privacy risks. Our work underlines the urgent need for stronger operating system controls and greater transparency in app data sharing practices, specifically, when the app is not visible to its users.

Acknowledgments

We are grateful to the anonymous PETS 2026 reviewers for their insightful comments, and guiding us in the final version of this paper. The third author is supported in part by an NSERC Discovery Grant.

References

[1] Android. Alarm Manager - Android background works. https://developer.android.com/develop/background-work/services/alarms/schedule. Accessed: 2024-10-18.

[2] Android. Android background execution limits - Android developers. https://developer.android.com/about/versions/oreo/background. Accessed: 2024-10-18.

[3] Android. Android handlers - Android developers. https://developer.android.com/reference/android/os/Handler. Accessed: 2024-08-21.

[4] Android. App hibernation - Android developers. https://developer.android.com/topic/performance/app-hibernation. Accessed: 2024-11-11.

[5] Android. App standby buckets - Android developers. https://developer.android.com/topic/performance/appstandby. Accessed: 2024-09-05.

[6] Android. Asynchronous works - Android background tasks. https://developer.android.com/develop/background-work/background-tasks/asynchronous. Accessed: 2024-10-18.

[7] Android. Background execution limits - Android developers. https://developer.android.com/about/versions/oreo/background. Accessed: 2024-11-03.

[8] Android. Background location limits - Android developers. https://developer.android.com/about/versions/oreo/background-location-limits. Accessed: 2024-11-03.

[9] Android. Behavior changes: Apps targeting Android 14 or higher - Android developers. https://developer.android.com/about/versions/14/behavior-changes-14. Accessed: 2024-11-11.

[10] Android. Behavior changes: Apps targeting Android 15 or higher - Android developers. https://developer.android.com/about/versions/15/behavior-changes-15. Accessed: 2024-11-11.

[11] Android. Broadcasts - Android background works. https://developer.android.com/develop/background-work/background-tasks/broadcasts. Accessed: 2024-10-18.

[12] Android. Dalvik executable format - AOSP. https://source.android.com/docs/core/runtime/dex-format. Accessed: 2024-11-08.

[13] Android. Executors - Android developers. https://developer.android.com/reference/java/util/concurrent/Executors. Accessed: 2024-09-14.

[14] Android. Implicit broadcast exceptions - Android developers. https://developer.android.com/develop/background-work/background-tasks/broadcasts/broadcast-exceptions. Accessed: 2024-10-08.

[15] Android. Instrumentationtestrunner - Android developers. https://developer.android.com/reference/android/test/InstrumentationTestRunner. Accessed: 2024-09-14.

[16] Android. Jobscheduler - Android background works. https://developer.android.com/reference/android/app/job/JobScheduler. Accessed: 2024-10-18.

[17] Android. Persistent works - Android background tasks. https://developer.android.com/develop/background-work/background-tasks/persistent. Accessed: 2024-10-18.

[18] Android. Privacy in Android 10 - Android developers. https://developer.android.com/about/versions/10/privacy. Accessed: 2024-11-11.

[19] Android. Privacy in Android 11 - Android developers. https://developer.android.com/about/versions/11/privacy. Accessed: 2024-11-11.

[20] Android. Processes and app lifecycle - Android developers. https://developer.android.com/guide/components/activities/process-lifecycle. Accessed: 2024-08-21.

[21] Android. Set important deprecation - jobinfo.builder. https://developer.android.com/reference/android/app/job/JobInfo.Builder#setImportantWhileForeground(boolean). Accessed: 2024-11-05.

[22] Android. Support for long-running workers - Android background works. https://developer.android.com/develop/background-work/background-tasks/persistent/how-to/long-running. Accessed: 2024-11-08.

[23] Android. User dismiss notification - foreground services. https://developer.android.com/develop/background-work/services/foreground-services?hl=en#user-dismiss-notification. Accessed: 2024-10-18.

[24] Android. Workmanager - Android background works. https://developer.android.com/reference/androidx/work/WorkManager. Accessed: 2024-10-18.

[25] APKPure. APKPure - Download APK on Android with free online APK downloader. https://apkpure.com/. Accessed: 2024-11-01.

[26] Xin Chen and Sencun Zhu. DroidJust: Automated functionality-aware privacy leakage analysis for Android applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec'15)*, June 2015.

[27] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, February 2017.

[28] Huajun Cui, Guozhu Meng, Yan Zhang, Weiping Wang, Dali Zhu, Ting Su, Xiaodong Zhang, and Yuejun Li. TraceDroid: A robust network traffic analysis framework for privacy leakage in Android apps. In *Science of Cyber Security (SciSec 2022)*, 2022.

[29] Xiaolin Du, Zhemin Yang, Jiapeng Lin, Yinzhi Cao, and Min Yang. Withdrawing is believing? detecting inconsistencies between withdrawal choices and third-party data collections in mobile apps. In *2024 IEEE Symposium on Security and Privacy (SP)*, May 2024.

[30] Electronic Frontier Foundation. apkeep - a command-line tool for downloading APK files from various sources. https://github.com/EFForg/apkeep/. Accessed: 2024-11-04.

[31] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *20th USENIX Security Symposium (USENIX Security 11)*, San Francisco, CA, August 2011. USENIX Association.

[32] FinancialPost.com. Double-double tracking: How Tim Hortons knows where you sleep, work and vacation. News article (June 12, 2020). https://financialpost.com/technology/tim-hortons-app-tracking-customers-intimate-data.

[33] Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern Paxson. An analysis of the privacy and security risks of android vpn permission-enabled apps. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 349–364, New York, NY, USA, 2016. Association for Computing Machinery.

[34] Javassist. Java bytecode manipulation simple - Javassist. https://www.javassist.org/. Accessed: 2024-11-14.

[35] John Wu. Magisk. https://github.com/topjohnwu/Magisk/releases. Accessed: 2024-11-08.

[36] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep Kumar Shukla. InviSeal: A stealthy dynamic analysis framework for Android systems. *Digital Threats: Research and Practice*, March 2023.

[37] Jianfeng Li, Hao Zhou, Shuohan Wu, Xiapu Luo, Ting Wang, Xian Zhan, and Xiaobo Ma. FOAP: Fine-Grained Open-World android app fingerprinting. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1579–1596, Boston, MA, August 2022. USENIX Association.

[38] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. You shall not repackage! demystifying anti-repackaging on android. *Computers & Security*, 103:102181, April 2021.

[39] mitmproxy. An interactive HTTPS proxy - mitmproxy. https://mitmproxy.org/. Accessed: 2024-11-05.

[40] Trung Tin Nguyen, Michael Backes, Ninja Marnau, and Ben Stock. Share first, ask later (or never?) studying violations of GDPR's explicit consent in android apps. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3667–3684. USENIX Association, August 2021.

[41] Ole André Vadla Ravnås. Frida. https://frida.re/. Accessed: 2024-11-05.

[42] Open source community. An xposed module that disables SSL certificate checking for the purposes of auditing an app with cert pinning. https://github.com/Fuzion24/JustTrustMe/. Accessed: 2024-11-09.

[43] Open source community. Aya - eBPF library for the Rust programming language. https://aya-rs.dev/. Accessed: 2024-11-05.

[44] Open source community. Bypass root check pro modern Xposed API module (with Java & native C/C++ hooks). https://github.com/gauravssnl/BypassRootCheckPro/. Accessed: 2024-11-09.

[45] Open source community. Community managed domain list. https://github.com/v2fly/domain-list-community. Accessed: 2024-11-11.

[46] Open source community. eBPF - Extended Berkeley Packet Filter. https://ebpf.io/. Accessed: 2024-11-05.

[47] Open source community. Lsposed framework. https://lsposed.org/. Accessed: 2024-11-04.

[48] Open source community. Simple to use root checking Android library and sample app. https://github.com/gauravssnl/BypassRootCheckPro/. Accessed: 2024-11-11.

[49] Sajjad Pourali, Nayanamana Samarasinghe, and Mohammad Mannan. Hidden in plain sight: Exploring encrypted channels in Android apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, November 2022.

[50] Sajjad Pourali, Xiufen Yu, Lianying Zhao, Mohammad Mannan, and Amr Youssef. Racing for TLS certificate validation: A hijacker's guide to the Android TLS galaxy. In *33rd USENIX Security Symposium (USENIX Security 24)*, August 2024.

[51] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the Android permissions system. In *28th USENIX Security Symposium (USENIX Security 19)*, August 2019.

[52] Julian Schutte, Dennis Titze, and J M De Fuentes. AppCaulk: Data leak prevention by injecting targeted taint tracking into Android apps. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (Trustcom'14)*, September 2014.

[53] Skylot. Jadx - Dex to Java decompiler. https://github.com/skylot/jadx. Accessed: 2024-11-08.

[54] Peter Späth. *Pro Android with Kotlin: Developing Modern Mobile Apps*. Apress, 2022.

[55] TechCrunch.com. Uber begins background collection of rider location data. News article (Nov. 28, 2016). https://techcrunch.com/2016/11/28/uber-background-location-data-collection/.

[56] The OW2 consortium. The asm library is a production-quality open source library for reading, writing, and manipulating jvm bytecode. https://asm.ow2.io/.

Accessed: 2024-11-14.
[57] TheHackerNews.com. How to stop Facebook app from tracking your location in the background. News article (Feb. 22, 2019). https://thehackernews.com/2019/02/facebook-location-tracking.html.
[58] Wired.com. Closing apps to save your battery only makes things worse. News article (March 15, 2016). https://www.wired.com/2016/03/closing-apps-save-battery-makes-things-worse.
[59] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *ACM Conference on Computer & Communications Security (ACM CCS'13)*, 2013.

Appendix

A Instrumenting Short-lived Processes, Concurrent Apps, and Java Interfaces

Short-lived and descendant processes. Android apps make extensive use of multi-threading, which adds complexity to tracing processes and accurately attributing network connections to background works. While we collect process and thread information for background activities, this alone does not fully address cases where app code dynamically spawns new processes or threads. Such processes, especially those initiated in native code, are challenging to trace reliably. Many of these are short-lived, created for minor actions like sending a single network packet, and can exist only momentarily, making effective tracing even more difficult.

Since Android's Linux kernel spawns processes through system calls like `fork` and `clone`, our initial approach was to hook these functions using Frida [41] and LSPosed [47]. However, this came with significant challenges. Frida requires a known process ID or the capability for self-spawning, which hindered rapid identification and attachment to newly executed background works. Attempts to pause and resume processes for Frida attachment often led to app crashes. Similarly, using LSPosed to hook the `fork` function introduced frequent stability issues. Moreover, FRIDA-based repackaging proved impractical at scale, introducing severe stability and DEX integrity issues[38].

To overcome these limitations and effectively track the chain of process, subprocess, and thread creations, we turned to kernel level instrumentation by utilizing eBPF's tracepoint program. After examining various tracepoints in Android, we identified that all process creations could be traced using the `task_newtask` and `sched_process_fork` tracepoints. This approach enables us to construct the process creation chain, proving robust against direct syscalls. It allows us to identify executed processes, their child processes, threads, and the responsible user ID. Consequently, we can attribute network connections to their originating processes, even in the context of short-lived, dynamically spawned threads.

Concurrent app instrumentation. We analyze multiple apps simultaneously (for scalability), necessitating the separation of their logs generated by the agent through API hooking, we append the stack trace, user ID, process ID, and thread ID by default to all records we generate. However, for network traffic, each network connection does not inherently carry an identifier specific to an app. Additionally, some apps are designed to be proxy-agnostic, meaning they do not use the proxy configurations set in the device settings for some connections [28, 49, 50]; instead, they establish direct connections to their intended destinations. To achieve traffic separation and redirection, we leverage a kernel level methodology based on eBPF as introduced in Marvin [50]. However, Marvin's approach does not fully meet our objectives, as it binds apps to destination domains. If two apps connect to the same destination, we cannot attribute the connection to the correct app. This limitation does not affect Marvin as it analyzes only one app at a time. However, we need to handle multiple apps potentially connecting to the same host, especially as many apps use popular third-party SDKs like Facebook. Moreover, we use the obtained data to identify the failed connections whose sockets were created by the kernel but not processed in the proxy for our measurements. To facilitate this, we additionally capture the source address and port of the connections, which are unique within the OS, to map them to the corresponding apps. However, accessing such information is not straightforward, as the cgroups used by Marvin do not provide source port and IP information because they operate before the socket is assigned in the OS. Therefore, we utilize kprobes in eBPF to hook the `tcp_connect` kernel function, which runs after the socket is assigned, and provides us this port/IP information. By combining cgroups and kprobes, we obtain the necessary information to attribute network traffic to apps and separate their traffic, even with multiple apps on the test device. We will publish a standalone version of the traffic separation and redirection component alongside Panoptes, as it can be used in other measurement studies.

Java interfaces/abstract class API hooking. Hooking Android's background work APIs, which are primarily interfaces, presents a significant challenge as interfaces lack concrete implementation.

To hook classes that implement these interfaces, we employed runtime analysis using Java reflection to identify the implementing classes and their methods. However, the LSPosed framework's classloader API is inadequate for retrieving class information. Instead, we directly load DEX files from the file system, using reflection to identify classes implementing our target interfaces. This method, while effective for statically loaded classes, fell short for dynamically loaded binaries commonly used by packers. To address this, we hooked `BaseDexClassLoader`, the fundamental class loader in Android runtime responsible for loading classes from DEX files. This allowed us to intercept and examine the process of loading classes at runtime, enabling us to identify, and subsequently hook, newly added classes that implement our target interfaces as they were dynamically loaded. This made Panoptes effective to track privacy exposure even from dynamically loaded code.

B Other Considerations

Changes in Android 14 and 15. We conducted our experiments on Android 13, as it had the highest usage rate when Panoptes development began in November 2023. However, since then, Android versions 14 and 15 have been released, and some devices have transitioned to them. We summarize the privacy-related restrictions in background works introduced by these versions[9, 10].

In Android 14, new features directly impact how apps handle background works and services. First, apps are required to specify at least one foreground service type for each foreground service in their manifest file. Second, runtime-registered BroadcastReceivers must declare their export status by specifying the appropriate flag (`RECEIVER_EXPORTED` or `RECEIVER_NOT_EXPORTED`) when registering the receiver. Additionally, JobScheduler jobs must complete

their operations within a few seconds [9]; otherwise, the system triggers an ANR (Application Not Responding) error. In earlier versions, such jobs would fail silently. Furthermore, there are new restrictions on starting activities from the background, requiring apps to explicitly opt in if their activities are intended to accept intents from background activities or other apps.

In Android 15, further restrictions on background work have been introduced. Foreground services of types `dataSync` and the new `mediaProcessing` are subject to a total runtime limit of six hours within a 24-hour period. If they exceed this limit, the system calls the service's `onTimeout` method, giving it a few seconds to call `stopSelf()`; failure to do so results in a system exception. Also, apps can no longer start certain types of foreground services from a `BOOT_COMPLETED` BroadcastReceiver, and attempting to do so throws a `ForegroundServiceStartNotAllowedException`. Moreover, apps holding the `SYSTEM_ALERT_WINDOW` permission must have a visible overlay window before starting a foreground service from the background; otherwise, starting such a service results in an exception [10].

Based on our understanding, these changes in Android 14 and 15 do not affect our analysis results, except for the scenario where a BroadcastReceiver using `BOOT_COMPLETED` starts a service in Android 15, which is now restricted. Panoptes operates within the app's context and does not rely on behaviors that have been restricted in these newer Android versions. The modifications primarily target long-running foreground services, background activity launches, and stricter permission requirements, which are not central to Panoptes's functionality.

App invisibility awareness. We noticed that 86.4% apps establish network connections while being minimized. To determine whether these apps are aware of their own visibility status, we hooked the `isInteractive()` method from the `PowerManager` class, which returns `true` when the device is awake and ready for user interaction. We found that 2049 apps dynamically called the `isInteractive()` API; notably, 2,015 of these apps invoked this method while in a minimized state, and 1,237 apps did so within background works (with some overlaps)–suggesting that many apps potentially check their visibility status both when minimized and during background execution.[11]

Lax app hibernation enforcement. Since Android 11, an app hibernation mechanism has been introduced to optimize performance and protect user data when an app remains unused for several months [4]. This mechanism revokes runtime permissions, and starting from Android 12, it also prevents the execution of jobs and alarms, blocks the receipt of push notifications, and clears the app's cached data. Android's definition of user interaction is broad, and includes: any engagement with the app, its widget, or notifications (excluding dismissible ones), and receiving an explicit intent broadcast from another app, using the app's services, or accessing an app's content providers, e.g., Input Method Editors (IMEs). We examined the app hibernation feature and identified discrepancies between Google's documentation [4] and the actual implementation. We developed a sample app designed to receive broadcasts and followed Android's guidelines to observe when the app transitioned into hibernation on a Pixel 7 device running Android 13. First, we checked AOSP configuration using the `device_config get permissions auto_revoke_unused_threshold_millis2` command and found that the precise duration required for an app to enter hibernation is three months. We further observed that not all runtime permissions are revoked when an app is hibernated; specifically, notification, physical activity, and nearby device permissions remain intact. Additionally, we found that apps in hibernation mode are still capable of receiving static broadcasts. Our findings suggest that the app hibernation feature is not entirely effective, as the continued receipt of static broadcasts allows for the execution of services, or even the main activity.

Effects of user interactions. The execution of background works can be dependent on user interactions with an app, such as using navigation in background, or initiating a media player. To understand the effects of such interactions, we conducted a small experiment using ThirdEye [49], which incorporates a keyword based UI interaction mechanism to automatically explore all app UI components. We used ThirdEye to analyze all apps that exclusively send private information using background works within the *User Assets* and *Location* types, resulting in 62 unique (data type, app) tuples across 56 apps, as some apps send multiple types of sensitive data. The objective was to determine whether such data is also exposed during the UI interactions and, if so, whether the exposure occurs within the same component or across different components (to determine if an exposure is due to background works or other app component). Through manual investigation, we identified that 43/62 (69.4%) exposures remained undetected by ThirdEye, due to background works not being triggering during the UI-based analysis. From the ones detected by ThirdEye, 14/19 exposures were established solely from the same component, indicating that background works were automatically triggered by UI interaction. 4 exposures were from different components than the background works, suggesting that while background works were not triggered, other components were responsible for sending the data. Additionally, in one instance, the exposures were from both background works and other components individually. Although our experiment is limited in scope here, it confirms that the majority of exposures in background works (69.4%) cannot be identified by existing tools, as the analysis duration is limited, and background works are often triggered by specific system/app conditions which may not be satisfied during the analysis.

[11] Note that `isInteractive()` is just one of several methods developers can use to determine an app's visibility or the device's state; other techniques include checking the app's lifecycle callbacks (e.g., `onPause()`, `onStop()`, `onResume()`), querying the `ActivityManager` for the app's process importance level, or using the `Visibility` API to detect whether the app's UI elements are currently visible to the user.

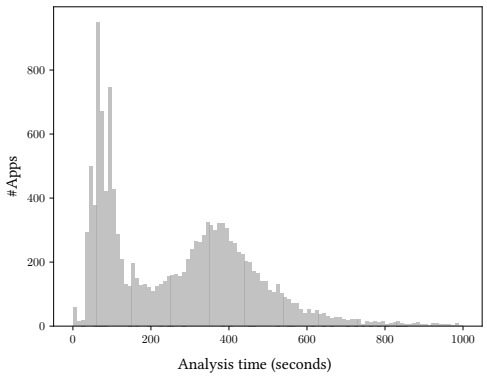

Figure 3: Distribution of app analysis times (108 apps over 1000 seconds).

	Type/Condition	Sensitive Transmission Information Categories				
		Device	Network	Location	User Assets	Any
JobScheduler	Network	5629	1477	18	14	5629
	Battery	18	0	2	0	18
	Charging	4	0	0	0	4
	Device Idle	88	8	0	0	88
	Storage	0	0	0	0	0
	Expedited	1	0	0	0	1
	Important	171	4	7	1	171
	Overall	5714	1479	22	14	5714
Broadcasts	Static	161	12	5	2	162
	Static (Undoc)	68	9	4	1	69
	Dynamic ⚠	532	69	23	6	533
	Overall	637	80	25	7	638
Services	Foreground	37	4	2	2	38
	Background ⚠	266	18	3	4	267
	Overall	302	22	5	6	304
	Overall	5842	1499	31	19	5843

Table 6: Transmission of sensitive information by background execution components categorized by type of exposure: Device, Network, Location, and User Assets. "Static (Undoc)" represents static undocumented broadcasts (see Sec. 5 under "Undocumented implicit broadcast exceptions" for more details). Rows marked with ⚠ indicate techniques not considered as a background work; thus, their values are excluded from the final overall row. Columns represent the categories of PII exposure, while rows specify the type of background operations. The reported results are not mutually exclusive as a single app may employ multiple background mechanisms and therefore contribute to several cells.

C Miscellaneous Tables, Figures and Listings

```
java.lang.Thread.getStackTrace(Thread.java:1841)
x0.c(Unknown Source:4)
M1.a(Unknown Source:245)
java.lang.reflect.Method.invoke(Native Method)
J.callback(Unknown Source:267)
LSPHooker_.connect(Unknown Source:18)
org.apache.http.conn.scheme.PlainSocketFactory.connectSocket(PlainSocketFactory.java:124)
...
com.gau.utils.net.connector.HttpConnector.run(HttpConnector.java:186)
a2.run(Unknown Source:13)
  java.lang.Thread.run(Thread.java:1012)
```

Listing 1: Stack trace of a TCP socket initiated from a BroadcastReceiver does not indicate the BroadcastReceiver as the origin but the Runnable class responsible for executing the work.

Parent \ Child	Service	JobScheduler	Broadcasts	Any
Service	14	23	3	38
JobScheduler	0	10	4	14
Broadcasts	2	17	3	22
Overall	16	48	10	72

Table 7: Nested Background works Executions Leading to Network Connections. Results are not mutually exclusive: a single app may appear in multiple columns when it utilizes different background works.

```
java.lang.Thread.getStackTrace(Thread.java:1841)
x0.c(Unknown Source:4)
M1.a(Unknown Source:245)
java.lang.reflect.Method.invoke(Native Method)
J.callback(Unknown Source:267)
LSPHooker_.connect(Unknown Source:18)
org.apache.http.conn.scheme.PlainSocketFactory.connectSocket(PlainSocketFactory.java:124)
...
com.gau.utils.net.connector.HttpConnector.run(HttpConnector.java:186)
a2.run(Unknown Source:13)
StackTracer_1hjXH6f6____13243_13556.run(StackTracer_1hjXH6f6____13243_13556.java:11)
java.lang.Thread.run(Thread.java:1012)
```

Listing 2: Stack trace injection result

```
java.lang.Thread.getStackTrace(Thread.java:1841)
....
java.lang.reflect.Method.invoke(Native Method)
J.callback(Unknown Source:138)
LSPHooker_.onReceive(Unknown Source:14)
android.app.ActivityThread.handleReceiver(ActivityThread.java:4306)
....
android.app.ActivityThread.main(ActivityThread.java:7918)
java.lang.reflect.Method.invoke(Native Method)
com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:548)
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:936)
```

Listing 3: Stack trace inside a onReceive method in the BroadcastReceiver class

```
java.lang.Thread.getStackTrace(Thread.java:1841)
....
java.lang.reflect.Method.invoke(Native Method)
J.callback(Unknown Source:138)
LSPHooker_com.app.view.receiver.WakeReceiver.onReceive(Unknown Source:14)
android.app.ActivityThread.handleReceiver(ActivityThread.java:4306)
....
android.app.ActivityThread.main(ActivityThread.java:7918)
java.lang.reflect.Method.invoke(Native Method)
com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:548)
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:936)
```

Listing 4: Stack trace inside a BroadcastReceiver

App Visible		App Minimized		Background Work		Overall	
Domain	# Apps	Domain	# Apps	Domain	# Apps	Domain	# Apps
firebaseinstallations.googleapis.com	9749	geller-pa.googleapis.com	3917	firebaselogging-pa.googleapis.com	4591	firebaseinstallations.googleapis.com	10316
firebase-settings.crashlytics.com	7429	graph.facebook.com	3733	firebaselogging.googleapis.com	3083	firebase-settings.crashlytics.com	7871
firebaseremoteconfig.googleapis.com	4092	crashlyticsreports-pa.googleapis.com	2852	crashlyticsreports-pa.googleapis.com	1978	geller-pa.googleapis.com	5091
graph.facebook.com	3716	firebase-settings.crashlytics.com	1414	geller-pa.googleapis.com	347	firebaseremoteconfig.googleapis.com	4886
fundingchoicesmessages.google.com	2070	digitalassetlinks.googleapis.com	1284	api.onesignal.com	96	firebaselogging-pa.googleapis.com	4705
googleads.g.doubleclick.net	2031	firebaseremoteconfig.googleapis.com	1154	firebaseremoteconfig.googleapis.com	89	crashlyticsreports-pa.googleapis.com	4614
geller-pa.googleapis.com	1809	googleads.g.doubleclick.net	1002	digitalassetlinks.googleapis.com	79	graph.facebook.com	4042
app.adjust.com	799	play.googleapis.com	835	tnc16-alisg.isnssdk.com	54	firebaselogging.googleapis.com	3160
ms.applovin.com	700	firebaseinstallations.googleapis.com	818	play.googleapis.com	54	googleads.g.doubleclick.net	2739
www.facebook.com	661	startup.mobile.yandex.net	603	s3.amazonaws.com	53	fundingchoicesmessages.google.com	2144
rt.applovin.com	655	api.onesignal.com	509	googleads.g.doubleclick.net	52	digitalassetlinks.googleapis.com	1862
d.applovin.com	619	ms.applovin.com	495	firebaseinstallations.googleapis.com	45	play.googleapis.com	1168
api.onesignal.com	529	rt.applovin.com	454	adsmetadata.startappservice.com	45	app.adjust.com	837
digitalassetlinks.googleapis.com	512	web.facebook.com	323	www.clarity.ms	40	ms.applovin.com	802
firebaseinappmessaging.googleapis.com	488	www.googleadservices.com	290	infoevent.startappservice.com	39	rt.applovin.com	770
startup.mobile.yandex.net	467	d.applovin.com	267	ws.batch.com	35	startup.mobile.yandex.net	744
mobile.yandexadexchange.net	452	pagead2.googlesyndication.com	262	wsmetrics.batch.com	34	www.facebook.com	744
api2.branch.io	436	firebaseremoteconfigrealtime.googleapis.com	232	sdk-api-v1.singular.net	33	d.applovin.com	721
cdn.branch.io	424	assets.adobedtm.com	180	info.startappservice.com	32	api.onesignal.com	580
crashlyticsreports-pa.googleapis.com	414	config.uca.cloud.unity3d.com	172	app.adjust.com	30	firebaseinappmessaging.googleapis.com	556
firebaseremoteconfigrealtime.googleapis.com	406	www.googleapis.com	166	combine.urbanairship.com	26	mobile.yandexadexchange.net	488

Table 8: Top 20 third-party domains contacted by apps. Red and blue shades denote Google and Facebook, respectively; unshaded domains represent other providers.