Tracking Privacy Risks In Android Applications

Purpose

This report explores privacy in Android mobile applications by examining how sensitive information can flow from a device to external services. Using the static analysis tool *Amandroid*, a small set of Android apps are analyzed to identify potential privacy leaks, such as device identifiers or location data being sent over the network. The goal of this study is twofold: to gain a deeper understanding of mobile application privacy concerns and to develop practical skills in empirical software analysis. The findings are classified and described to underscore common patterns and risks in app data handling.

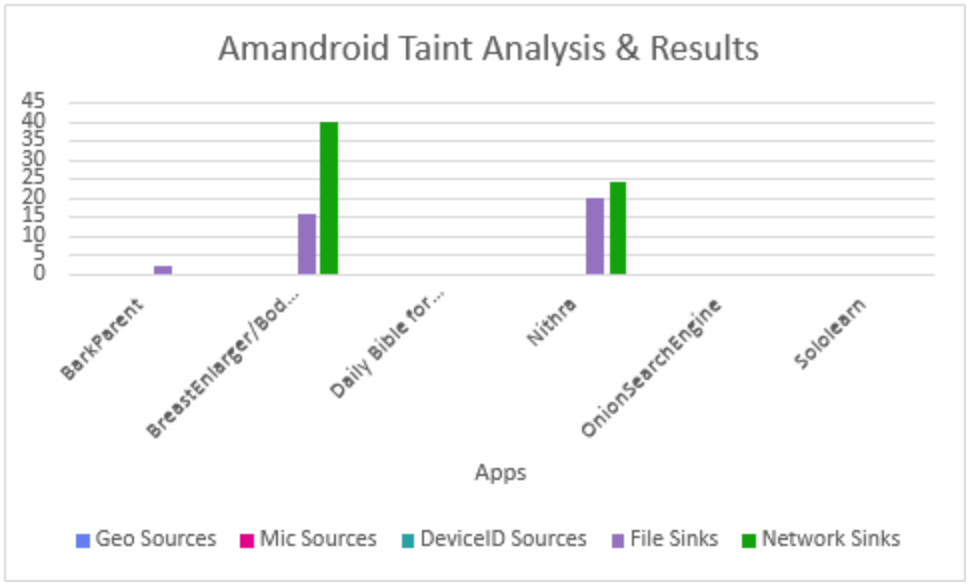
Running Amandroid & Its Results

Running Amandroid on the apps in my folder was a process that spanned two days, hampered to an extent by resource constraints. Out of the ten apps selected, I was able to complete the analysis successfully for six. The remaining four apps were too large to process fully on my laptop, resulting in errors and failures to properly save the results.

The larger apps caused Java heap errors due to insufficient allocated memory. To mitigate this, the page space was increased to 2.4 GB and the Java heap size was modified to 2 GB, which is the upper limit of what the laptop could handle. This allowed for smoother execution and helped prevent hangups during the analysis for the smaller apps. Even with these adjustments, the larger apps still struggled to run completely -- an indication that the resource demands exceeded the laptop's capabilities. Given that more than half of the applications ran successfully, the analysis proceeded despite initial challenges.

Below is both a table and a graph detailing the identification of taint paths

App	Geo Sources	Mic Sources	DeviceID Sources	File Sinks	Network Sinks
BarkParent	0	0	0	2	0
BreastEnlarger/BodyEditor/PhotoEditor	0	0	0	16	40
Daily Bible for Women	0	0	0	0	0
Nithra	0	0	0	20	24
OnionSearchEngine	0	0	0	0	0
Sololearn	0	0	0	0	0



Notes: All zeros are for sources are consistent with the apps' privacy expectations or with the implementation choices that avoid exposing sensitive data through detectable API calls. File sinks appear in apps that store user-generated content or cache locally. Network sinks appear in apps that transmit analytics or upload data. Where a privacy policy/EULA is available, it helps explain why taint analysis detects zero sources or the presence of sinks. As a result, the data paths between sources and sinks could not be established in these apps. However, the absence of detected sources does not necessarily mean the app collects no sensitive data; it only means that the data flows from standard Android APIs (Geo, Mic, DeviceID) to sinks were not detectable by Amandroid.

A Python script was used to collect the sources and sinks to compile the table and graph shown above. The script used is as follows:

```
import os
import re
import pandas as pd

BASE_DIR = r"C:\Users\jaade\OneDrive\Desktop\results"

GEO_KEYWORDS = ["android/location/LocationManager", "getLastKnownLocation",
"getLatitude", "getLongitude"]
MIC_KEYWORDS = ["android/media/AudioRecord", "MediaRecorder", "startRecording"]
DEVICE_ID_KEYWORDS = ["android/telephony/TelephonyManager", "getDeviceId",
"getSubscriberId", "android_id"]

FILE_KEYWORDS = ["java/io/FileOutputStream", "openFileOutput",
"java/io/FileWriter", "putString", "write"]
NETWORK_KEYWORDS = ["HttpURLConnection", "Socket", "okhttp3"]

def categorize_source(signature):
    sig = signature.lower()
    if any(k.lower() in sig for k in GEO_KEYWORDS):
        return "Geo"
    elif any(k.lower() in sig for k in MIC_KEYWORDS):
        return "Mic"
    elif any(k.lower() in sig for k in DEVICE_ID_KEYWORDS):
        return "DeviceID"
    return None

def categorize_sink(signature):
    sig = signature.lower()
    if any(k.lower() in sig for k in FILE_KEYWORDS):
        return "File"
    elif any(k.lower() in sig for k in NETWORK_KEYWORDS):
        return "Network"
    return None

summary_rows = []

for file_name in os.listdir(BASE_DIR):
    if file_name.endswith(".txt"):
        app_name = os.path.splitext(file_name)[0]
        geo_count = mic_count = device_count = 0
        file_count = network_count = 0

        with open(os.path.join(BASE_DIR, file_name), "r", encoding="utf-8") as f:
```

```
            inside_taint_path = False
            for line in f:
                line = line.strip()

                if line.startswith("TaintPath:"):
                    inside_taint_path = True
                    continue
                if inside_taint_path and not line:
                    inside_taint_path = False
                    continue

                if inside_taint_path:
                    matches = re.findall(r"@signature `([^`]*)`", line)
                    for sig in matches:
                        src_cat = categorize_source(sig)
                        if src_cat == "Geo":
                            geo_count += 1
                        elif src_cat == "Mic":
                            mic_count += 1
                        elif src_cat == "DeviceID":
                            device_count += 1

                        sink_cat = categorize_sink(sig)
                        if sink_cat == "File":
                            file_count += 1
                        elif sink_cat == "Network":
                            network_count += 1

        summary_rows.append({
            "App": app_name,
            "Geo Sources": geo_count,
            "Mic Sources": mic_count,
            "DeviceID Sources": device_count,
            "File Sinks": file_count,
            "Network Sinks": network_count
        })

df = pd.DataFrame(summary_rows)
df.to_csv("Amandroid_Taint_Summary.csv", index=False)
print("Summary table saved to Amandroid_Taint_Summary.csv")
```

The script checks each node signature against predefined keyword lists that correspond to the defined categories. For example, the Geo source list includes "*android/location/LocationManager*", "*getLastKnownLocation*", "*getLatitude*", and "*getLongitude*"; Mic sources include "*android/media/AudioRecord*", "*MediaRecorder*", and "*startRecording*"; Device ID sources include "*android/telephony/TelephonyManager*", "*getDeviceId*", "*getSubscriberId*", and "*android_id*". Similarly, File sinks include "*java/io/FileOutputStream*", "*openFileOutput*", "*java/io/FileWriter*", "*putString*", and "*write",* while Network sinks include *"HttpURLConnection", "Socket",* and *"okhttp3".* Whenever a node in the Amandroid output matched one of these keywords, it was automatically assigned to the corresponding category, which formed the basis for the table and graph. In instances where Amandroid failed to detect flows — such as network activity in Sololearn and Onion Search Engine — manual inspection of the decompiled source code was used to identify additional Network sinks, accounting for obfuscated code.

End User License Agreements & Privacy Policies

Briefly examined are two apps – BarkParent and Sololearn -- to see what their EULAs and privacy policies entail.

BarkParent collects parent and child account details, device-usage information, monitored content (such as messages and media), location data, and technical/usage logs needed to perform parental-monitoring services. The company processes this data to analyze activity, generate alerts, provide reports to parents, and operate its platform, and use vetted third-party service providers for hosting and processing. Bark states that identifiable user data is generally retained for about 30 days, with some information potentially kept longer for reporting or legal obligations. The company does not sell or rent personal data for marketing but may share information with service providers or disclose it when required by law.

Sololearn collects registration information (such as name, email, age), learning activity, user-generated content, device and usage analytics, cookies/tracking data, and—when using AI features—any text provided to its AI tools, which may be processed externally (e.g., by OpenAI). Sololearn uses this data to operate and improve its platform, personalize learning, support advertising through aggregated demographic insights, and send marketing communications, and shares information with analytics, hosting, and payment providers. Data is retained while the account is active, and after deletion some information may remain for a "reasonable period" for business, legal, or research needs. Sololearn states that it does not sell personally identifiable data but does share non-identifiable aggregated data with advertisers and transfers personal data to third-party processors as needed to run the service.

App Behavior & How This Fits Into Taint Analysis

Analyzing the apps while in use allows us to see firsthand what personal information, if any, is gathered without having to sign up or sign in. Barkparent and Sololearn, the two apps previously discussed in regards to their privacy policies and EULA above, required log in to use. Using Onion Search Engine app didn't ask for any personal data upon use, and fully functioned without requiring an account. It also didn't request any personal data or permissions, such as location, contacts, or storage. According to its Privacy Policy, the service operates under a No-Log Policy and does not collect IP addresses, search queries, or browsing history. The policy also specifies that the app uses no cookies, tracking scripts, or intrusive JavaScript. The EULA confirms that the app is provided under a privacy-focused license, with no provisions that allow collection of personal data for basic search functionality. Optional services, like Onion Drive, may collect information if a user creates an account, but these were not used during normal app

operation. Overall, the app's behavior aligns with its Privacy Policy and EULA, showing no unjustified data collection or privacy violations.

The Terms of Service and Privacy Policy of the Daily Bible for Women app clearly spell out what users are consenting to, including the collection of certain personal data. For instance, according to its App Store listing, the app may collect contact information (like email address), identifiers (such as a user ID or device ID), usage data (how you interact with the app), and diagnostics (crash reports or performance data). Because these categories are explicitly stated in the policy, the app's collection of such data is not a privacy violation, as it falls within the user's informed consent. Even if the data is more technical (like device identifiers or usage patterns) rather than deeply personal, it is still covered under the agreed terms. Since users agree to the EULA/terms before using the app, this practice is legally and ethically permitted under its own policies.

The Nithra Resume Builder app does not have a dedicated, app-specific EULA or Terms of Service, which makes it less clear exactly what legal rights and obligations the developer claims for this app. However, the Privacy Policy is detailed, explaining what personal and non-personal data may be collected, how it is used, and under what circumstances. The policy emphasizes transparency in the use and storage of data and clearly states that personal information is not sold or rented. If the app's actual behavior matches what is described in the Privacy Policy, there is no indication of a privacy violation. The lack of a dedicated terms document is a minor limitation, but overall, the app demonstrates responsible handling of user data and respects user privacy.

Finally, the Body Editor – Breast Enlarger / Photo Editor app is a photo-editing tool that allows users to modify body and facial features in images. According to its privacy policy, the app does not collect personally identifiable information beyond what the user explicitly provides, such as images selected for editing. It uses facial and body recognition to detect landmarks for editing, but this data is not stored and is deleted immediately after processing if temporarily uploaded to servers. The app collects non-identifiable usage data for analytics in aggregated form, and all data transmissions are encrypted. It does not sell or share personal data with third parties for profiling or advertising. Overall, the app demonstrates transparent handling of user data, limits storage to what is strictly necessary for editing, and respects user privacy while providing AI-powered photo-editing features.

Further Analyzing the Taint Analysis

The earlier reporting of the Amandroid Taint Analysis results combined with observing the apps' behavior in use brings things full circle. Comparing the two confirms the consistency between what the apps claim to collect and what their code actually handles. None of the applications revealed sensitive sources such as geolocation, microphone input, or device identifiers, which indicates that they are not gathering high-risk personal data behind the scenes. The taint paths that were detected primarily involved file sinks for Body Editor, BarkParent, and Nithra Resume Builder—each of which aligns with the apps' expected functionality, such as saving edited images or storing user-created documents. Because these sinks relate to data the user intentionally provides, they do not constitute a privacy violation and fit within the permissions and behaviors described in each app's privacy policy.

Network sinks were identified in the PhotoEditor and Nithra apps, suggesting that some data is transmitted over the internet. Based on their publicly stated privacy practices, this network activity is consistent with features such as cloud-based photo processing or sending anonymized analytics information. Importantly, Amandroid detected no taint paths leading from sensitive sources to network sinks, meaning no evidence of personal data being improperly transmitted. Overall, the taint analysis makes it relatively easy to map each path to its corresponding app feature, and none of the detected flows indicate unjustified or undisclosed data collection.

Analyzing App Source Code

Two apps, Sololearn and Onion Search Engine – which during the initial taint analysis detected nor sinks or sources -- were decompiled to analyze their source code for network flows.

Sololearn: Although Amandroid did not detect any network sinks for this APK, manual analysis of the decompiled source code revealed clear network behavior. Specifically, the app invokes *accept()* on a ServerSocket instance (e.g., *this.d.accept()*), which waits for and establishes incoming network connections. Once a connection is accepted, data can be exchanged using the socket's input and output streams.

This constitutes a network flow because the application receives or transmits data over a network socket. The reason Amandroid did not flag this before during the taint analysis is likely because the network operations occur inside obfuscated third-party library code, which static taint analysis tools often cannot fully parse or classify.

Further analysis revealed a similar circumstance -- Amandroid did not automatically detect network sinks for Sololearn, but manual analysis of the decompiled source code revealed a direct network flow. Inside a method within the Fabric/Crashlytics networking component (*io.fabric.sdk.android.services.network.HttpRequest…*), the code calls *url.openConnection()* and casts the result to *HttpURLConnection*.

This indicates that the application (or included SDK) is establishing an HTTP connection to a remote server. Any data written to this connection's output stream or read from its input stream constitutes network communication.

Onion Search Engine: In the analyzed APK, the method *d.a.a.a.a.e.g.a(URL, Proxy)* creates an outbound network connection. It calls *url.openConnection(proxy)* and casts the result to *HttpURLConnection*, establishing a connection to a remote server, optionally through a proxy. The returned *HttpURLConnection* can be used to send or receive data over the network, making this a clear network flow. Additionally, the APK revealed a network flow in the Google Mobile Ads SDK. The method *com.google.android.gms.ads.internal.gmsg.HttpClient.zza(HttpClient$b)* calls *getOutputStream()* on a network connection object. Data written to this output stream is sent over the network to a remote server, representing an outbound network flow. This flow was not detected by Amandroid, likely due to obfuscation and abstraction layers in the SDK, but manual inspection via decompiling confirms that network communication occurs here.

Acknowledgement

I computed the SHA-256 hash of my last name plus the current year in lowercase using PowerShell. The command used was:

```
$bytes = [System.Text.Encoding]::UTF8.GetBytes("adesalu2025")
$hash = [System.Security.Cryptography.SHA256]::Create().ComputeHash($bytes)
($hash | ForEach-Object { $_.ToString("x2") }) -join ""
```

The resulting hash was:

4c5fd914e99d35f5ca6b84b1bc9c235981f69140df63fb1650542bc0873ac08f

The first hexadecimal digit -- 4 -- determined that I should use the dataset mp4-appset-4.zip for analysis.