Detecting Privacy Non-Compliance in Wearable Apps via Knowledge Graphs and LLMs

Tran Thanh Lam Nguyen
University of Insubria, Varese, Italy
ttlnguyen@uninsubria.it

Barbara Carminati
University of Insubria, Varese, Italy
barbara.carminati@uninsubria.it

Elena Ferrari
University of Insubria, Varese, Italy
elena.ferrari@uninsubria.it

***Abstract*—Wearable devices are becoming increasingly popular in modern life, making significant contributions to human health monitoring. While security and privacy violations in standard apps have been extensively studied in many previous work, wearable apps have received comparatively little attention. This paper presents an automated framework that leverages Large Language Models (LLM) to identify privacy violations in Android wearable apps. The method evaluates both declared practices by extracting third-party services and shared data types from a Knowledge graph generated from the Manifest and Data Safety sections, and actual behaviors by analyzing sent-out network traffic. We evaluated the proposal on 711 popular companion apps and found that 67.5% violate the declared data collection and sharing practices, with 4.8% leaking data to undeclared third-party services.**

***Index Terms*—Wearable apps, Privacy, Large Language Models (LLM), GraphRAG**

I. Introduction

Along with smartphones, wearable devices (e.g., smartwatches) and their app ecosystems are today an indispensable part of modern life.[1] Given their widespread adoption and access to sensitive personal data, it is therefore essential to assess the security and privacy risks associated with these apps. In this paper, we focus on investigating how wearable apps manage and share the sensitive data collected from end users.

Wearable apps can be categorized into three types: embedded, companion, and standalone [1]. Embedded apps run on the wearable's firmware or OS (e.g., heart rate monitor) and rely on companion apps on smartphones for full functionality, such as data synchronization via Bluetooth. In contrast, standalone apps work independently on the wearable device but require WIFI/LTE (with high battery cost) to send data online.[2] Since our study centers on data sharing, we focus on companion apps, particularly Android apps, which account for 71.75% of the mobile OS market.[3]

In general, companion apps may transmit sensitive data to their backend servers for processing or storage, and to third-party services (TPSs) (e.g., ads, analytics, or health platforms). In both cases, privacy regulations such as the GDPR (General Data Protection Regulation) and the CCPA (California Consumer Privacy Act) require apps to clearly disclose their data collection and sharing practices and to obtain user consent. To comply with this requirement, Google requires that app developers prepare the `Manifest` file, declaring the permissions requested by the apps and providing details on potential third-party sharing. In particular, the `Manifest` includes the names of TPSs but does not specify the type of data being shared. These are instead provided in the `Data Safety`[4] section on the Google Play console, which developers fill before app release. During app installation, users are actively asked for consent to the permissions declared in the `Manifest`, while no details about the specific TPS sharing are disclosed to them. Moreover, since `Manifest` and `Data Safety` are self-reported by developers without Google's verification, no mechanism currently ensures app compliance with the declared practices. Thus, in this paper, we aim to investigate the *companion app's compliance with the data sharing practices declared in the Manifest and Data Safety.*

In the literature, this problem has been investigated along two directions. The first focuses on risk related to the misalignment of permissions stated between the companion and embedded/standalone apps. For instance, [2], [3] have shown that an attacker can exploit this vulnerability to request permissions to collect sensitive data on one device (e.g., a smartwatch) and then leak the sensitive data to another device (e.g., a smartphone). The second direction focuses on assessing the privacy compliance of wearable apps by comparing the collection of sensitive data with what developers have declared, which is also the main objective of this paper. However, previous studies have focused on standard Android apps rather than wearable apps. In addition, existing solutions primarily employ traditional analysis methods, including static, dynamic, and hybrid approaches, which exhibit limitations in scalability and effectiveness. A notable example is [4] that employed a hybrid analysis to assess privacy violations that occur when standard Android apps integrate with the Facebook SDK. Specifically, using static analysis, they identify the presence of the Facebook SDK in the app. Next, with dynamic analysis and the aid of a man-in-the-middle (MITM) proxy, they record the type of data transmitted by the app through the Facebook SDK. The results showed that the Facebook SDK illegally collected three types of sensitive data, namely location, device ID/model, and the MAC address of the Wi-Fi router. Another

[1] https://scoop.market.us/smart-wearables-statistics/
[2] Google recommends minimizing data exchange in standalone apps.
[3] https://gs.statcounter.com/os-market-share/mobile/worldwide
[4] https://developer.android.com/google/play/integrity/other

979-8-3503-9281-4/25/$31.00 ©2025 IEEE

example is TraceDroid [5], which utilizes dynamic analysis to monitor privacy leaks in Android apps. However, it focuses only on three types of sensitive data: device information, location, and network details. Similarly, PTPDroid [6] relies solely on static analysis; therefore, it cannot provide an accurate observation of the app's actual runtime behavior, which can lead to false positives. In [7], we introduced Metaleak, which employs a MITM proxy to capture outgoing traffic from standard Android apps, with a focus on detecting privacy violations in image sharing through EXIF metadata. More recently, [8] evaluated data sharing from Android apps to TPSs using an MITM proxy and ChatGPT to identify data recipients. However, their dynamic analysis only examines device ID, location, and network, and moreover, observed only two specific classes for network connections, namely *com.android.okhttp* and *com.my.tracker*, so potentially overlooking others. In summary, existing solutions are primarily based on traditional analysis methods and focus solely on a limited set of sensitive data types. Moreover, an app may integrate with multiple TPSs simultaneously, an aspect that has not been addressed by previous proposals. Additionally, sensitive data can be collected not only by TPSs but also by the app's backend, which is an aspect that should not be overlooked. Furthermore, traditional approaches may provide high accuracy but are not scalable.

To address these limitations, we propose an automated method to identify all TPSs integrated into a companion app and to assess privacy violations arising from the sharing of sensitive data with both TPSs and the app's backend, considering 14 types of sensitive data defined in Google's documentation.[5] Specifically, we focus on evaluating two factors: (1) whether a companion app violates privacy compliance by sending, in its sent-out traffic, sensitive data that is not listed in the Data Safety; and (2) whether the destination of the app's sent-out traffic complies with the TPS configuration in its Manifest.

In particular, given a target app X, we analyze the sent-out traffic generated by X to determine the types of sensitive data it transmits over the Internet and to which destinations. This represents what we call the X's **observed behavior**. We also analyze the Manifest/Data Safety declarations to determine X's **theoretical behavior**. The aim is to assess whether the observed behavior diverges from the theoretical one.

However, building companion apps' theoretical and observed behaviors poses non-trivial challenges. Indeed, it is impossible to develop a straightforward strategy for parsing the Manifest to extract the theoretical behavior, due to the heterogeneity in the way app developers define the Manifest, which leads to using different XML tags even for the same purpose. To overcome this limitation, we propose utilizing Large Language Models (LLM) to represent the information contained in the app's Manifest and Data Safety as a Knowledge Graph (KG). This representation allows us to capture the complex relationships between TPSs declared in the Manifest and data types specified in Data Safety. We then again use LLM, enhanced with a Graph-based Retrieval-Augmented Generation (GraphRAG) approach, to reason over the obtained KG and accurately retrieve the types of shared data and the integrated TPSs. Similarly, to generate the observed behavior, we must process unstructured and heterogeneous sent-out traffic (i.e., traffic payload) and HTTP headers (i.e., traffic destination), where sensitive data may appear in multiple formats or encodings, and the traffic destination can be TPS or the app's backend. Additionally, deep links[6] can also serve as a channel for sending out sensitive data, often beyond the user's awareness [9]. LLM are employed again to interpret these data patterns and identify sensitive data and its destination. This is achieved by augmenting the LLM prompt with the definition of types of sensitive data extracted from the official Google documentation.

We conduct several experiments to test the effectiveness of our approach. The results show that the proposed LLM-based strategies are a valid option for modeling both theoretical and observed behaviors. We also evaluated our approach on 711 popular companion apps and found that 480 of them are not compliant with their declared Data Safety information. Specifically, 211 of them send sensitive data (e.g., User ID) not reported in the Data Safety to declared TPSs, meaning these apps violate Data Safety but comply with the TPS declaration. In contrast, 23 apps transmit sensitive data to undeclared TPS (i.e., through deep links), thereby violating both Data Safety and the TPS declaration. The remaining 246 apps only share sensitive data with their backend. In contrast, among the 231 apps that comply with Data Safety declarations, the majority (227 apps) also respect the declared data destinations in their Manifest files. However, a small portion (4 apps) transmit data to undeclared TPSs via deep links.

The remainder of this paper is organized as follows. Section II presents our LLM-based approach for modeling theoretical behavior, whereas Section III describes prompt engineering for observed behavior. Section IV reports experiments, while Section V concludes the paper.

II. Theoretical behavior

The Manifest is an XML file designed to allow app developers to list permissions, identify the app type, configure the TPS that the app integrates with, and provide several other relevant information. The Manifest often contains thousands of lines with many XML tags and attributes. Google provides a large number of XML tags to configure app attributes in the Manifest. However, it does not enforce strict syntax and structural checks, resulting in a lack of standardization and making it difficult to understand the Manifest's contents. As an example, to declare app type, Google recommends `<meta-data>` with attribute com.google.android.wearable.standalone.[7] However, our analysis of 5,000 wearable apps[8] reveals that 98% do not

[5]Source code is publicly available on GitHub https://github.com/research-mobile-security/WearLeak

[6]Deep links are URLs embedded directly into the app's source code used to integrate TPSs with the app without requiring declaration in the Manifest.

[7]https://developer.android.com/training/wearables/apps/standalone-apps

[8]We analyze our dataset by simply parsing the Manifest.

adhere to this guideline. This also applies to TPSs that are often specified inconsistently. For example, we found apps that, within the same Manifest, use different tags for each TPS they integrate with, sometimes using the `<service>` tag as well as the `<provider>` tag. In short, due to the heterogeneity in how Manifest files are created by app developers, even for the same purpose, and the unpredictability of the presence of the necessary tags, it is unfeasible to build a simple strategy for parsing the manifest file to determine TPSs.

To overcome this difficulty, given a target app X, we represent information contained in its Manifest and Data Safety as a Knowledge Graph (KG), called *ComplianceKG*$_X$. In general, a KG is a structured representation of information where concepts are modeled as entities (nodes) connected by relationships (edges), thus enabling reasoning over complex, interconnected data. However, since the presence of XML tags in the manifest is unpredictable, we cannot define a unique *ComplianceKG*$_X$ representation that works for all possible apps' manifests. To address this issue, we leverage LLM to generate the *ComplianceKG*$_X$ from X's Manifest and Data Safety. Once generated, the *ComplianceKG*$_X$ can then be analyzed to retrieve relevant information, such as the TPSs connected to the app X and the types of data X shares. To perform such analysis, we leverage LLM again to take advantage of its ability to reason over implicit relationships that might be difficult to express with traditional query languages. To further enhance its ability to interpret tag meanings and accurately retrieve information from *ComplianceKG*$_X$, we provide LLM with additional contextual information. In particular, we collect Google documentation that describes all XML tags used in Manifest files and augment the LLM prompt using GraphRAG (Graph-based Retrieval-Augmented Generation) [10], as described in the following.

A. GraphRAG

Retrieval-Augmented Generation (RAG) [11] extends LLM's capabilities by enabling access to specific domain knowledge bases without requiring retraining of the model for that domain. RAG typically involves three main steps: (1) embedding external authoritative data and storing it in a vector database; (2) embedding the user prompt and retrieving the most similar vectors; and (3) combining the prompt with retrieved data to create an augmented prompt. In our scenario, to augment the LLM prompt, we consider both Google's documentation as well as *ComplianceKG*$_X$ as an authoritative external source. Indeed, *ComplianceKG*$_X$ encodes the developer's self-declared app behavior (i.e., the Manifest and the Data Safety), which represents the most authoritative available description of the app's intended data practices (i.e., theoretical behavior). However, due to the large size of the manifest file and the limited input token capacity of LLM, passing the whole *ComplianceKG*$_X$ to the LLM is not optimal in terms of both performance and cost. Furthermore, not all XML tags in the Manifest are relevant to TPS integrations, and including too much redundant information in the input could lead to hallucinations and decreased accuracy. However, by treating *ComplianceKG*$_X$ as an external resource, we can leverage the RAG (step 2) and the Google Documentation to selectively extract only the subset of graph content related to our prompt (i.e., TPS sharing). When applying RAG, we must also consider that traditional RAG struggles to capture the relationships between pieces of information, resulting in poor performance when reasoning over interconnected data. While it works well with structured sources like Google's documentation, it proves ineffective with KGs [12] (i.e., *ComplianceKG*$_X$). This is mainly because the embedding vectors used by RAG are unstructured data, unable to represent the relationships between data points (i.e., structured data as KG).[9] Therefore, relying only on vector similarity and ignoring the relationships between XML tags will not provide sufficient context for LLM and will reduce the accuracy of their responses. To overcome this limitation, we adopt GraphRAG [10], as it can retrieve external data through both the similarity vector (i.e., similar to RAG) and the relationships in the KG. Similarly to RAG, the GraphRAG workflow consists of three steps, which are described in what follows.

1) Preparation of External Data source: GraphRAG uses both GraphDB (for *ComplianceKG*$_X$) and VectorDB (for Google's documentation) to store external data.

GraphDB - ComplianceKG$_X$. Given a target app X, we first extract relevant information from its Manifest and Data Safety, then we build a prompt that guides the LLM in generating *ComplianceKG*$_X$.

(a) Manifest tags extraction. We extract from X's Manifest all its XML elements, denoted as X_{Tags}. Then, for each element $t_j \in X_{\text{Tags}}$, we extract all its attributes' names, denoted as A_{t_j}. The set of all extracted attributes for all the elements in X_{Tags} is denoted as $X_{\text{TagAttributes}}$. Moreover, for each element $t_j \in X_{\text{Tags}}$, we also define the set of its relations with its attributes, denoted as r_{t_j}. The set of all relations for all the tags in X_{Tags} is denoted as $X_{\text{Relations}}$, where each element is formed by concatenating the string *"has_config"* with a tag t_j.

Example 1: Let us consider an app X with the following simple Manifest containing only two XML tags:

```
<manifest package="ai.coachify.coachify">
<service ns0:name="com.facebook.FacebookActivity"/>
    <intent-filter ns0:name="android.intent.
    action.VIEW"/>
    <intent-filter ns0:scheme="facebook.connect"/>
</service>
```

$X_{\text{Tags}} = \{\text{service}, \text{intent-filter}\}$. Moreover, tag `<service>` has only one attribute (i.e., ns0:name), so $A_{\text{service}} = \{\text{ns0:name}\}$, while for tag `<intent-filter>`, $A_{\text{intent-filter}} = \{\text{ns0:name}, \text{ns0:scheme}\}$. $X_{\text{TagAttributes}}$ contains A_{service} and $A_{\text{intent-filter}}$. Finally, $X_{\text{Relations}} = \{\text{has\_config\_service}, \text{has\_config\_intent\_filter}\}$.

(b) Data Safety extraction. We recall that the Data Safety lists data types that the app will collect and share. Thus, we

[9]For example, to identify TPS, RAG could search for the `<service>` tag (which is correct according to Google's documentation); however, the `<intent-filter>` and `<meta-data>` tags contained within the `<service>` tag could also provide this information (cf. Example 1).

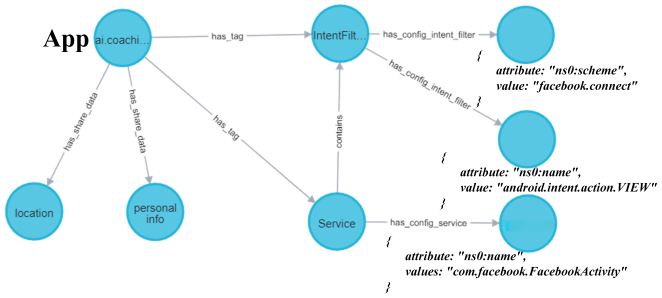

Fig. 1. Example of *ComplianceKG*$_X$

TABLE I
LLM PROMPTS

Prompt for *ComplianceKG*$_X$ generation
Given app X defined by a package name $\{package\_name\}$ and with the following Manifest tag information: $\{t_1\}$ is configured with attribute $\{A_{t_1}\}$ and $\{t_1\}$ is linked to $\{A_{t_1}\}$ through the relation $\{r_{t_1}\}$; $\{t_2\}$ is configured with attribute $\{A_{t_2}\}$ and $\{t_2\}$ is linked to $\{A_{t_2}\}$ through the relation $\{r_{t_2}\}$; ... In addition, app X is also declared with Data Safety information, including $\{X_{DataShare}\}$ and X linked to $\{X_{DataShare}\}$ through relation $\{r_{sharedata}\}$.
GraphRAG retrieval: prompt $p1$ (for VectorDB)
Identify the XML tags used to configure the appropriate third-party service.
GraphRAG retrieval: prompt $p2$ (for GraphDB)
For an app with package name $\{package\_name\}$ and a manifest represented as a knowledge graph, identify the subgraph containing information about shared data and integrated third-party services, if any, knowing that the following XML tags $\{TagDescriptor_{TPS}\}$ are used to identify third-party services.
Final prompt for TPSs and data types retrieval
Identify the list of third-party services and the types of data shared by the app with package name $\{package\_name\}$, given that the app's manifest is represented as a knowledge graph as follows: *ComplianceKG_TPS*$_X$
Sensitive data type identification in sent-out traffic
You are provided the app's sent-out traffic payload in $\{traffic_{sent\text{-}out}\}$ and the HTTP header in $\{http_{header}\}$. Your task is to analyze the outgoing network traffic from an Android app, determine whether the data is sensitive, and identify the destination URL of the sent traffic. Knowing that the following examples provide categories and details of sensitive data in Android: $\{\langle SensitiveCategory_1, \{DataTypes_1 \dots DataTypes_n\}\rangle, \dots, \langle SensitiveCategory_{14}, \{DataTypes_1 \dots DataTypes_m\}\rangle\}$.

extract this list, denoted as $X_{\text{DataShare}}$, by crawling X's Data Safety from the Google Play Store. We also add a new relation $r_{sharedata}$ = *"has_share_data"* in $X_{\text{Relations}}$ to link X with $X_{\text{DataShare}}$.

(c) Prompt generation for ComplianceKG$_X$. Given a target app X, we instantiate a prompt using the template in Table I -1st row, where variables in curly braces $\{\}$ are filled with X's specific values. Then, we pass the obtained prompt to *LLMGraphTransformer()*[10] method provided by Langchain to generate the KG, which is finally stored in GraphDB.

Example 2: Let us consider the Manifest provided in Example 1 and suppose that $X_{\text{DataShare}} = \{\text{location}, \text{personal info}\}$. *ComplianceKG*$_X$ returned by *LLMGraphTransformer()* is represented in Figure 1.

VectorDB - Google documentation. We use the Google documentation describing the syntax, structure, and attributes of XML tags defined in Manifest file. Specifically, for each tag t_i described in the Google documentation, we create a tuple, denoted $TagDescriptor_{t_i} = \langle syntax_{t_i}, contained\_in_{t_i}, description_{t_i}, attributes_{t_i}\rangle$, where $syntax_{t_i}$ contains information about the syntax of t_i, $description_{t_i}$ provides information about the meaning, function, purpose of usage, how it works, and the appropriate context in which t_i should be used, $contained\_in_{t_i}$ provides information about the position where the tag t_i appears, which helps to determine the valid context in which t_i is allowed to appear. For example, the `<meta-data>` tag is only valid when it is placed inside tags such as `<activity>`, `<application>`, `<service>`, etc. Through $contained\_in_{t_i}$, GraphRAG can infer neighboring tags related to t_i in the KG. Finally, $attributes_{t_i}$ lists the valid attributes that t_i can have.

B. Hybrid retrieval & Augmenting LLM Prompt

GraphRAG supports a two-step hybrid retrieval process. The first step queries VectorDB to collect XML tags that are relevant only for TPS management. The initial prompt of the first step, p_1 in Table I, is encoded in an embedding vector v_{p_1}, used to query those $TagDescriptor$ in VectorDB that are relevant for TPS integration (aka whose embedding vectors are similar to v_{p_1}). The result is a list of embedding vectors, denoted by $\hat{v}_{p_1}$. The $\hat{v}_{p_1}$ is then decoded back into text form (i.e., the $TagDescriptor_{TPS}$). Note that this process is performed only once, as prompt p_1 does not change with different apps.

The second step aims to extract the portion of *ComplianceKG*$_X$ related to TPS configuration. Thanks to $TagDescriptor_{TPS}$, we provide additional information that helps LLM identify not only XML tags related to TPSs but also its child tags. This is important because manifests are heterogeneous, so developers can configure TPSs in child tags even though Google specifies that the parent tag is the one that should be used for that purpose. For instance, in Example 1, the developer used both the parent (`<service>`) and child tags (`<intent-filter>`) to configure integration with Facebook. The prompt of the second step, p_2 in Table I, is sent to LLM via the *graph.query()*[11] method provided by Langchain to query GraphDB. We therefore obtain *ComplianceKG_TPS*$_X$, which is the portion of *ComplianceKG*$_X$ that contains data sharing information and tags used to configure TPSs.

Finally, we send the final prompt (4th row in Table I) to LLM to determine the list of TPSs the app integrates with and the types of data the app shares.

III. Observed Behavior

A wearable app's sent-out traffic contains various types of unpredictable information, not necessarily sensitive data (e.g., timestamp), encoded in different formats, such as JSON, XML, or key-value. Additionally, the transmitted values can

[10]https://python.langchain.com/v0.1/docs/use_cases/graph/constructing/

[11]https://python.langchain.com/docs/tutorials/graph/

vary depending on the format, communication protocol, and regulations of the app developer.[12]

Determining the destination of sent-out traffic is also a challenge because the app operates as a black-box. Thus, the communication protocols used by apps to establish internet connections with TPS and its backend are unpredictable, such as HTTP, HTTPS, gRPC, and WebSocket. Moreover, the Data Safety section only indicates what sensitive information the app will share, but does not specify the destination of this information. In contrast, the Manifest file is used to configure the TPS that the app integrates with, but it cannot determine whether data is sent from the app to the TPS. Indeed, when an app integrates with a TPS, we could have: (1) bidirectional communication involving both incoming and outgoing traffic, or (2) unidirectional communication, where the app only receives traffic from the TPS for its function.

As such, developing a simple strategy to parse sent-out traffic that fits all possible companion apps is challenging. To overcome this, we again leverage LLM to analyze the app's sent-out traffic (i.e., HTTP header and related payload) to detect sensitive data and the destination of the traffic. For this purpose, we first need to determine what types of data should be considered sensitive. In this paper, we assume that all data types that require explicit Android permissions to access (e.g., location data regulated by the *ACCESS_COARSE_LOCATION* permission) are sensitive. Therefore, to create a list of sensitive data types, we first refer to Google's official documentation on permissions.[13] However, we found that this documentation is not sufficient in some specific cases. For example, Google's documentation describes the *READ_CONTACTS* permission in a rather vague way, as: *"allows an application to read the user's contacts data"*, where this could be interpreted as phone numbers and/or email addresses. To overcome this shortcoming, we also use Google's privacy documentation.[14] Based on Google's documentation on permissions and privacy, we define 14 sensitive data categories, modeled as pairs $\langle \text{SensitiveCategory}, \text{DataTypes} \rangle$. As an example of some of the identified 14 categories, Table II reports the data types detected during our experiments.

The information about sensitive data categories is combined with the sent-out traffic in the prompt to assist the LLM in identifying the types of sensitive data. Secondly, we also attach the HTTP header to the prompt to determine the recipient URL, which helps us identify the traffic's destination (i.e., TPS or the app's backend). Finally, we submit to the LLM a prompt created following the template in Table I, 5th row, enhanced with the identified sensitive data types, a variable $\text{traffic}_{\text{sent-out}}$ containing the app's sent-out traffic content, and a variable $\text{http}_{\text{header}}$ containing HTTP headers. The resulting output is a list of sensitive data (if any) and the corresponding destination URL.

[12]As an example, if an app sends the password, say "Password@11235", to register, the value captured in sent-out traffic could be "Password%4011235" (that is, the "@" character is converted to "%40") if the app uses the key-value format, while the password value remains the same if the app uses the JSON structure. As another example, although we input the value "male" for gender in the profile, the app sends the value "1" (i.e., the developer defines "male" to have a value of 1).

[13]https://developer.android.com/reference/android/Manifest.permission

[14]https://developer.android.com/privacy-and-security/declare-data-use

IV. Experiments

We run experiments to assess the accuracy of LLM-based approaches in generating apps' theoretical and observed behaviors, and to evaluate their effectiveness in detecting non-compliant apps. We first introduce the dataset used.

A. Dataset

We downloaded 5,000 apps supporting smartwatches from the Google Play Store in Europe between January and March 2025. Among these, we selected the 1,000 most popular based on their installation counts. Then, we installed and ran each of them on a smartphone (model Samsung Galaxy M51) and examined the results. If an app can be successfully installed and executed, we classify it as a companion app, since standalone and embedded apps can only operate on a smartwatch. We obtained a set of 711 companion apps, denoted as $\mathcal{D}_{\text{evaluation}}$. For each of these apps, we collected the corresponding Manifest and Data Safety. To capture the traffic sent by apps, which is essential for modeling their observed behaviors, we leverage the MetaLeak framework [7]. This framework incorporates a MITM proxy and a Capture Traffic (CT) module that can classify the app's incoming and outgoing traffic. In particular, for each app, we utilize MetaLeak to capture its outgoing traffic by selecting only HTTP POST and PUT requests, as these methods are typically used to transmit user data to external servers. The captured traffic is then stored in a text file, named Sent-Out Traffic File (SOTF), that includes both the HTTP headers and the payload content. However, MetaLeak requires manual interaction with the app's UI to generate traffic. To automate this process, we combined MetaLeak with Droidbot,[15] a tool for testing input generation for Android. Specifically, each companion has been automatically used for 3 minutes. Additionally, for apps that require registration/login to access deeper functionalities, we manually perform the registration/login process before using Droidbot. We also pre-define a set of information used for registration/login, for example: {*email:* `androidtest@gmail.com`*, password: "Password@11235", birthday: "1992-10-21", gender: "male", height: "170", weight: "80"*} to label which data is entered by the user into the app, facilitating sent-out traffic analysis, especially in case of detecting apps sharing usernames and passwords in plain text (cf. Section IV-C).

B. Theoretical and observed models' validation

We rely on LLM to generate both the app's theoretical and observed behaviors. Like other ML algorithms, LLM could suffer from misclassification (e.g., identifying the wrong shared data type) and/or hallucination (e.g., indicating a TPS service not specified in the Manifest). To test whether the LLM-based strategies are a valid option for modeling theoretical and observed behaviors, we test their precision.

[15]https://github.com/honeynet/droidbot

In particular, we randomly select 100 apps from $\mathcal{D}_{\text{evaluation}}$, denoted as $\text{Model}_{\text{val}}$, and manually inspect them to determine their behaviors, as described in the following.

1) **Theoretical model validation:** For each app in $\text{Model}_{\text{val}}$, we manually review its safety information published on the Google Play Store to determine the list of shared data types. We then compare this list with the one returned by LLM and we verify that the two lists are the same for each app in $\text{Model}_{\text{val}}$. This comparison confirms that *the LLM strategy detects 100% of shared data types*.

Similarly, we manually check the list of TPSs integrated with apps in $\text{Model}_{\text{val}}$. Specifically, we rely on the XML tags in the Google documentation used to configure TPSs (i.e., $TagDescriptor_{TPS}$) and manually inspect these tags in the apps' Manifests. We also review their child tags to ensure we do not miss TPS information. Then, we store the TPS names obtained through the manual process in a set associated with the app, say X, denoted as TPS_manual_X. This set is then compared with the results returned by LLM (denoted as TPS_LLM_X). If (1) TPS_manual_X =TPS_LLM_X, then the LLM returns the correct results for the app X; if (2) $\mid TPS\_manual_X$ - $TPS\_LLM_X \mid > 0$, LLM misses some of the TPSs manually identified; finally, if (3) $\mid TPS\_LLM_X$ - $TPS\_manual_X \mid > 0$, LLM has misclassified some TPSs. For the apps in $\text{Model}_{\text{val}}$, we find that 93% are in case (1), 0% in case (2), and 7% in case (3). We further analyze the 7% misclassified and find that misclassification was due to TPS name inconsistencies, especially for those provided by Google and Facebook. For example, the LLM splits Google Firebase (i.e., official TPS's name) into two distinct TPSs, namely Firebase Analytics and Firebase Messaging, because Google Firebase provides both features. A similar issue happens with Facebook, where the TPS name is sometimes labeled as Meta. These are not serious misclassifications and can be corrected using simple normalization techniques.

2) **Observed model validation:** First, we manually analyze the sent-out traffic payload of each app X in $\text{Model}_{\text{val}}$, and determine its list of shared data types.[16] Given an app X, the results of the manual analysis of its sent-out traffic is a set $Dt\_manual_X \subseteq \{Dt_1, Dt_2, \ldots, Dt_{14}\}$, where Dt_j indicates the sensitive data categories corresponding to the data types found in its traffic. If the sent-out traffic does not contain any of the data types from the 14 sensitive data categories, we set Dt_manual_X to an empty set. Then, for each app, we input its SOTF content into the LLM prompt described in Section III (model GPT-4o). Similarly to the manual verification, we store the returned sensitive data categories in Dt_LLM_X. To estimate the accuracy, we consider the following metrics: *True Positive* (TP), that denotes the number of apps that actually sent out some sensitive data, and the list of data types detected by LLM matches the one verified manually, i.e., $Dt\_manual_X = Dt\_LLM_X$; *True Negative* (TN), that represents those apps where both LLM and manual verification determined that they did not send out sensitive data, i.e., $Dt\_manual_X = Dt\_LLM_X = \emptyset$; *False Positive* (FP), that denotes the number of apps that did not actually send sensitive data, but LLM sent-out analysis identified some data types, $Dt\_manual_X = \emptyset \wedge Dt\_LLM_X \neq \emptyset$; finally *False Negative* (FN), that represents apps that sent some sensitive data, but LLM fails to identify them or misclassifies the leaked data into the wrong category, i.e., $Dt\_manual_X \neq Dt\_LLM_X$. Based on TP, TN, FP, and FN, the resulting precision, recall, and F1-score are 0.91, 0.83, and 0.87, respectively. The LLM strategy achieves high precision (91%), indicating that most of the predicted sent-out sensitive data are correct. However, the recall is slightly lower (83%), meaning some actual data types were missed. The F1-score (87%) demonstrates that our approach achieves a good balance, though recall could be improved by reducing false negatives.

To verify whether LLM captures all destination URLs, we manually analyze all HTTP headers contained in sent-out traffic of each app X in $\text{Model}_{\text{val}}$. Depending on the type of protocol used, we determine the destination URL through different parameters in the HTTP header, for example, *"host"* for the HTTPS protocol and *"authority"* for the GRPC protocol. To identify the owner of the destination URL (denoted as $\text{Destination\_URL}_{\text{manual}}$), we employ a script built on Tracker Radar.[17] This script also enables us to determine whether the traffic is directed to a TPS or the app's backend. Next, we compare the results of our manual inspection with the destination URLs returned by LLM (i.e., the result of the prompt in row 5th of Table I). Similarly, we identify the corresponding owners (denoted as $\text{Destination\_URL}_{\text{LLM}}$) and compare them with $\text{Destination\_URL}_{\text{manual}}$. We find that $\text{Destination\_URL}_{\text{LLM}}$ = $\text{Destination\_URL}_{\text{manual}}$, for all the analyzed apps, meaning that LLM can detect 100% of the destinations from the HTTP header.

C. Non-compliance detection

In this experiment, we compare the theoretical and observed behaviors of the apps in our dataset to detect non-compliant apps. In particular, we check: (1) whether the apps send sensitive data that is not listed in Data Safety; and (2) whether the destination of the app's sent-out traffic complies with the TPS configuration in its Manifest.

1) Data Safety compliance. We find that 480/711 (≈67.51%) apps violate privacy for at least one sensitive data type. For these apps, we find in their observed behaviors one or more types of sensitive data that were not declared in their theoretical behaviors. Table II shows the distribution of violated data categories and data types, together with the corresponding number of apps. The remaining 231/711 (≈32.49%) apps comply with privacy regulations for the types of sensitive data shared.

2) Destination compliance. To provide comprehensive results, we separate the analysis between apps that comply with Data Safety and those that do not.

[16]Initially, we search for known DataTypes as keywords, but we find that developers often use alternative terms (e.g., "Lat" for latitude, "LastUpdate" for timestamp). To avoid possible errors, we manually review all outgoing traffic.

[17]https://github.com/duckduckgo/tracker-radar

TABLE II
SENSITIVE DATA TYPES & CORRESPONDING PRIVACY VIOLATION DISTRIBUTION

Sensitive Category	Data Type	Number of apps
Device or other IDs	IMEI, MAC address, Widevine Device ID, Firebase Installation ID, Advertising ID, IP Address, Google Advertising ID	297/711 (≈41.77%)
Personal info	full name, username, password, email address, user ID, phone number, birthday, gender	227/711 (≈31.93%)
Location	longitude, latitude, altitude	64/711 (≈9%)
Health & Fitness	weight, height, medical records (heart rate, etc.), exercise (step, swim, etc.)	20/711 (≈2.81%)
App info & performance	crash & app logs, CPU/RAM/battery	15/711 (≈2.11%)
Messages	email/SMS/MMS/chat (subject line, sender, recipients)	4/711 (≈0.56%)
Financial info	accounts/credit card number, transaction history	2/711 (≈0.28%)

Apps violating Data Safety (480 apps). We find that 211/480 (≈43.96%) of the apps send information used to track users and provide advertising, such as User ID and Google Advertising ID (GAID), to TPSs declared in the Manifest. This indicates that while these apps violate privacy regulations regarding the types of sensitive data shared (i.e., Data Safety), they still comply with the declared TPS integration in the Manifest. However, 23/480 (≈4.8%) apps send sensitive information, including User ID, GAID, and the app's log, to TPSs not explicitly declared in the Manifest through deep links, to track user behavior, app performance, and advertising. This reveals a serious violation of privacy with respect to both the types of data shared and their transmission destinations. The remaining 246/480 (≈51.24%) apps only send sensitive data to the app's backend, indicating that these apps only violate Data Safety policies, but do not share sensitive data with TPSs.

Apps not violating Data Safety (231 apps). 227/231 (≈98.27%) of the apps send data only to TPSs declared in the Manifest. This indicates that these apps comply with both Data Safety and the declared destinations. However, 4/231 (≈1.73%) apps send data to TPSs not explicitly declared in the Manifest through deep links. This shows that although these apps comply with Data Safety, they do not comply with the declared destinations.

3) Users' credentials handling. In addition to non-compliance detection, we conduct a further analysis to investigate how apps handle user credentials. We observe that 132/711 (≈18.57%) of the apps send username and password in plaintext during the registration/login process. Although apps may use HTTPS to encrypt data during transmission, storing user account information in plaintext poses a significant security risk, as it allows developers to gain complete control over the user's account. [18] Furthermore, the failure to encrypt users' passwords also violates GDPR Article 32 – security of processing.[19]

V. CONCLUSION & FUTURE WORK

In this paper, we introduce a novel approach to detect privacy non-compliance in the wearable app ecosystem. Our proposed framework is a combination of hybrid analysis and LLM (along with context-enhancing methods, including GraphRAG and prompt engineering), making it unconstrained by any specific type of sensitive data. As future work, we plan to develop the proposed framework into an AI Agent embedded directly on smartphones and wearable devices.

VI. ACKNOWLEDGEMENTS

This work was supported in part by project SERICS (PE00000014) under the NRRP MUR program, funded by the EU-NGEU. The authors' views are their own and do not necessarily reflect those of the EU or Italian MUR.

REFERENCES

[1] B. Olabenjo and D. Makaroff, "Information leakage in wearable applications," in *Security, Privacy, and Anonymity in Computation, Communication, and Storage: 12th International Conference, SpaCCS 2019, Atlanta, GA, USA, July 14–17, 2019, Proceedings 12*. Springer, 2019, pp. 211–224.

[2] D. Yeke, M. Ibrahim, G. S. Tuncay, H. Farrukh, A. Imran, A. Bianchi, and Z. B. Celik, "Wear's my data? understanding the cross-device runtime permission model in wearables," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 2404–2421.

[3] M. Tileria, J. Blasco, and G. Suarez-Tangil, "{WearFlow}: Expanding information flow analysis to companion apps in wear {OS}," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 63–75.

[4] D. Rodriguez, J. A. Calandrino, J. M. Del Alamo, and N. Sadeh, "Privacy settings of third-party libraries in android apps: A study of facebook sdks," *Proceedings on Privacy Enhancing Technologies*, 2025.

[5] H. Cui, G. Meng, Y. Zhang, W. Wang, D. Zhu, T. Su, X. Zhang, and Y. Li, "Tracedroid: A robust network traffic analysis framework for privacy leakage in android apps," in *International Conference on Science of Cyber Security*. Springer, 2022, pp. 541–556.

[6] Z. Tan and W. Song, "Ptpdroid: Detecting violated user privacy disclosures to third-parties of android apps," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 473–485.

[7] T. T. L. Nguyen, B. Carminati, and E. Ferrari, "Metaleak: Assessing image metadata leakage in android apps," in *2024 IEEE/ACS 21st International Conference on Computer Systems and Applications (AICCSA)*, 2024, pp. 1–10.

[8] D. Rodriguez, J. M. Del Alamo, C. Fernández-Aller, and N. Sadeh, "Sharing is not always caring: Delving into personal data transfer compliance in android apps," *IEEE Access*, vol. 12, pp. 5256–5269, 2024.

[9] H. Hu, H. Wang, R. Dong, X. Chen, and C. Chen, "Enhancing gui exploration coverage of android apps with deep link-integrated monkey," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 6, pp. 1–31, 2024.

[10] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, D. Metropolitansky, R. O. Ness, and J. Larson, "From local to global: A graph rag approach to query-focused summarization," *arXiv preprint arXiv:2404.16130*, 2024.

[11] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A survey on rag meeting llms: Towards retrieval-augmented large language models," in *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*, 2024, pp. 6491–6501.

[12] H. Han, H. Shomer, Y. Wang, Y. Lei, K. Guo, Z. Hua, B. Long, H. Liu, and J. Tang, "Rag vs. graphrag: A systematic evaluation and key insights," *arXiv preprint arXiv:2502.11371*, 2025.

[18] This security risk is listed in the Common Weakness Enumeration – CWE-256: Plaintext Storage of a Password) https://cwe.mitre.org/data/definitions/256.html

[19] https://gdpr-info.eu/art-32-gdpr/