Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

Privacy Compliance Analysis in Mobile and Wearable Applications

by
Tran Thanh Lam Nguyen

Supervised by
Prof. Barbara Carminati
Prof. Elena Ferrari

A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science in the Department of Theoretical and Applied Sciences at the University of Insubria

Varese, Italy
October 2025

Declaration of Authorship

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except if otherwise specified in the text. However, every single bit of it could never be achieved without the support of so many people, for whom I shall always remain grateful.

Date	Tran Thanh Lam Nguyen

Dedication

To my mother and father, who have always supported and helped me through the most difficult times, especially during my first experience living and working abroad, far away from family.

To my younger brother, a doctor, who has taken my place in caring for our parents during my absence and who has also supported me with health matters.

To my parents-in-law, who have always shown care and offered their help whenever I returned home.

To my uncles, aunts, and cousins (on my mother's side), who have shared my difficulties, supported me, and prepared the hometown delicacies I love whenever I came back home.

To my family,
I express my deepest gratitude to my wife, a strong pillar, a compassionate companion who has understood and shared my struggles throughout my PhD journey. I believe she has sacrificed greatly so that I could fulfill my dream. Moreover, I feel fortunate that my wife, a Master's graduate in Data Science, could listen to and understand my research stories while also inspiring me in choosing the technologies applied in this very dissertation.

To myself,
My PhD journey began with a promise I made to my wife, and today I can proudly say: "I have kept my word to you". Beyond that, the PhD has been a trial I set for myself. I did not begin as someone extraordinary, nor as someone meticulously prepared during my Bachelor's or Master's years. Instead, it was a way to answer the question: "Am I truly good enough to face this challenge after having achieved my initial success in an industrial career?" In the end, I have come to realize that you never truly know when you are ready or good enough, but you can always choose courage, as in the famous quote: "Courage isn't having the strength to go on; it is going on when you don't have the strength".

Acknowledgments

To complete this dissertation, my own efforts alone would not have been sufficient. Every form of support, whether direct or indirect, from my professors, familly, colleagues, and friends has been invaluable and deserves a place of honor not only in this dissertation but also in my academic career as a whole, both now and in the future. I would like to express my deepest gratitude to all of them.

First of all, I firmly believe that *"ability is of little account without opportunity"*. I owe my profound gratitude to Professor Elena Ferrari and Professor Barbara Carminati for the trust they placed in me and for the golden opportunity they offered me to study and conduct research at STRICT SociaLab, University of Insubria. I am convinced that without the opportunity provided by them, I would have remained an engineer in Vietnam and my research career would never have taken shape. During my three years at STRICT SociaLab, I received dedicated supervision, rigorous scientific feedback, and invaluable research experience from both Professors. Moreover, they helped me greatly in my personal life during the very first days of my arrival in Italy. I could not have completed this dissertation on time without their timely support. I do not know what else to say except: *"Thank you for everything"*. I sincerely hope to continue collaborating with Professors even after I am no longer a doctoral student at STRICT SociaLab. In addition, I would like to express my sincere gratitude to Professors Claudio Agostino Ardagna and Meng Li for their thoughtful evaluations and constructive feedback on my thesis. Their comments greatly helped me refine and strengthen the quality of this work.

Secondly, I would like to extend my special thanks to Dr. Ha Xuan Son (RMIT University). Dr. Son introduced me to STRICT SociaLab, which laid the foundation for the beginning of my PhD journey. Furthermore, he was my close collaborator and co-author on my very first publications, which gave me the confidence to embark on this long path. As the Buddha once said, *"Our encounters in life are the result of karmic connections"*, I truly hope that we will continue to be collaborators in science in the future.

During my time at STRICT SociaLab, I had the opportunity to meet many friends, researchers, and colleagues who supported me and enriched this experience, because a PhD is not only about research but also about life in its many colors. I would like to thank Dr. Giorgia Sirigu, the first person to welcome me at the university. Her energy and knowledge strengthened the positive atmosphere of the lab. In addition, I am grateful to Dr. Jesus Fernando Cevallos Moreno, whose shared his life experience in

Italy helped me tremendously. I will never forget his assistance and the positive energy he brought into my life. I would also like to express my thanks to Dr. Desiree Manicardi for the kindness and warmth she consistently showed me. I also thank Khaoula Hidawi for the interactions we had during my time in Italy.

Finally, I would like to express my gratitude to Mrs. Roberta Viola, Mr. Mauro Santabarbara, and the university's administrative offices for their constant support with technical and administrative matters.

Last but not least, my parents, family, and friends have also played an essential role in helping me complete this work. Although they may not have fully understood what I was doing, their willingness to share, listen, and be present eased my pressure and gave me calmness in my work. I am deeply grateful to my parents, my younger brother, and my wife, who have always supported me and provided the peace of mind I needed to fully dedicate myself to this endeavor. I am also thankful to all my teachers in Vietnam, who laid the academic foundation that enabled me to complete this dissertation.

Abstract

In recent years, smartphones and wearable devices have been the two most dynamic ecosystems, with billions of users and millions of applications driving their growth. Indeed, according to Datareportal, as of July 2025, there are 7.4 billion smartphones in use globally, while wearable devices have reached 600 million units according to Statista. Furthermore, there are 8.93 million applications (aka, apps) released worldwide, with 3.553 million apps in the Google Play Store and 1.642 million in the Apple App Store, as reported by Bankmycell. On average, each user installs more than 40 apps on their device.

However, the growth of these two ecosystems is built on a trade-off in user privacy, as 65.83% of the ecosystem's revenue comes from advertising. This raises concerns about the serious invasion of users' privacy as app developers and hackers continuously exploit their sensitive information for revenue reasons. Although the European Union and the USA have enacted laws to protect privacy, such as the General Data Protection Regulation (GDPR) or the California Consumer Privacy Act (CCPA), that require apps to notify users and obtain explicit consent before collecting and processing sensitive data, violations remain widespread and have become increasingly sophisticated. Specifically, these violations can take advantage of users' common smartphone usage habits and the weaknesses of smartphone operating systems (OS), especially the Android OS.

Indeed, in this thesis, we first introduce a novel attack vector that demonstrates how sharing images containing sensitive metadata can unintentionally or intentionally lead to the leakage of users' personal or confidential information. To validate our finding and assess its prevalence, we use traditional analysis. While the results confirm that this newly discovered attack vector has a significant impact, they also highlight the inherent limitations of traditional analysis. Therefore, we propose a new solution based on Large Language Models (LLM) to build an early-warning system capable of detecting potential leaks of sensitive metadata embedded in images. Our evaluation, conducted on datasets from traditional analysis, shows highly promising results.

Finally, we develop our LLM-based solution toward a more general framework by assessing privacy non-compliance in wearable apps. Specifically, we evaluate whether these apps respect users' privacy in sharing sensitive data and its destinations across 14 sensitive categories as defined by Google.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

The modern world is witnessing an explosion of mobile devices, particularly smartphones and wearable devices. Indeed, it is undeniable that the invention of mobile devices has revolutionized human lifestyle forever because of the convenience they deliver. Hidden in a compact size is a powerful processing capacity capable of meeting all complex human requirements, from communication and work to photography, shopping, entertainment, and healthcare. In fact, statistics from *gs.statcounter.com*[1] (June 2025) show that smartphones account for 64.35% of the market while desktops and laptops account for only 35.65%, showing a significant shift in demand from personal computers to mobile devices. Additionally, statistics from *soax.com*[2] (July 2025) indicate that 64.35% of global web traffic comes from mobile devices, while a survey by *explodingtopics.com*[3] (June 2025) states that, on average, people spend 4 hours and 37 minutes using their phones every day. In summary, all the statistics suggest that mobile devices are an indispensable part of human life.

Contributing to the success of mobile devices cannot fail to mention the operating system (OS) and applications (apps), which are considered the *"soul"* of creating an incredibly diverse mobile ecosystem. As of July 2025, Android OS accounts for 71.85%[4], completely dominating the smartphone OS market. Moreover, Wear OS, a minimalist version of Android specifically for wearable devices (e.g., smartwatches), accounts for 27%[5] market share in the global smartwatch market, second only to Apple's watchOS. Indeed, Android OS and WearOS attract device manufacturers and app developers because of their license-free and freedom of customization compared to their direct competitors (iOS and watchOS), which are Apple's closed ecosystem. However, the nature of open-source OS is also a double-edged sword, leading to the Android OS and Wear OS being more frequent targets of security attacks and user privacy violations compared to iOS and watchOS [24]. Therefore, *this thesis mainly focuses on the Android OS and*

[1] https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide
[2] https://soax.com/research/mobile-website-traffic
[3] https://explodingtopics.com/blog/smartphone-usage-stats
[4] https://www.demandsage.com/android-statistics/
[5] https://www.counterpointresearch.com/insights/global-smartwatch-market-2024/

the Wear OS.

Indeed, in the first quarter of 2025, Kaspersky Security Network[6] recorded 12 million attacks on mobile devices involving malware, adware, or unwanted apps. Additionally, 180,000 malicious installation packages were identified. On the other hand, statistics on the mobile app market's revenue reveal alarming facts. Specifically, in the total revenue of $522.7 billion for the entire mobile app market in 2024, revenue from advertising is $344.10 billion (65.83%), in-app purchases[7] are $172.50 billion (33%), and paid apps reach only $6.10 billion (1.17%). The above statistics indicate that advertising is the primary driving force behind maintaining the mobile app ecosystem, as most apps are provided for free[8]. However, *"there is no such thing as a free lunch"*; users are trading their privacy to use free apps without being fully aware of the risks associated with this issue. Specifically, apps often silently collect sensitive information from users to provide advertising campaigns suitable for each user's profile [80].

On May 25, 2018, the European Union (EU) adopted the General Data Protection Regulation (GDPR)[9] - a full set of laws that govern how apps and websites must handle the collection, processing, sharing, and storage of users' data. The GDPR is based on two main principles: (1) lawfulness: personal data can only be collected with the user's explicit permission, and (2) transparency: users must be clearly informed why their data is being collected and how it will be used and shared. Similarly, the California Consumer Privacy Act (CCPA),[10] enacted in 2018, states GDPR-like provisions to protect personal data of California residents, regardless of where the business processing the data is located (inside or outside the USA). However, GDPR and similar regulations are not enough to ensure users' privacy and security for two main reasons. Firstly, most app developers ignore those regulations and keep collecting users' personal information for profit [75], even though they could face fines of up to €20 million or 4% of a company's global annual turnover (depending on the severity of the violation). Secondly, hackers can exploit security vulnerabilities that still exist in the OS to bypass the protection mechanism and access sensitive data.

For the reasons mentioned above, research on **privacy compliance within the mobile and wearable device ecosystem** has become a significant concern in the academic community and is also the primary focus of this thesis. Literature reviews show that there are countless attack scenarios targeting mobile and wearable devices; however, three traditional methods are used to analyze these risks, namely, static analysis, dynamic analysis, and hybrid analysis [62]. Specifically, static analysis [19] investigates the app's source code to identify possible risks without executing the app. For example, this method focuses on control-flow, data-flow, and API calls to determine execution paths, predict the values of variables at specific program points, and identify destinations that receive sensitive data. In contrast, dynamic analysis [19] assesses the app's observed behavior at run-time to determine potential attacks or malware, for example, by exam-

[6] https://securelist.com/malware-report-q1-2025-mobile-statistics/116676/
[7] The app is free but users have to pay to use some exclusive features.
[8] https://meetanshi.com/blog/mobile-app-download-statistics/
[9] https://gdpr-info.eu/
[10] https://oag.ca.gov/privacy/ccpa

ining the presence of sensitive data in the app's sent-out traffic and system metrics such as CPU, RAM, and battery usage. Finally, hybrid analysis combines static and dynamic analysis to leverage the advantages of both methods.

Although traditional analysis methods have been widely used in many security and privacy research [15, 59, 62, 74], these methods also have many unresolved weaknesses. First, because static analysis does not provide real-time observation of app security and privacy violations, it can lead to a high false positive rate [62]. Specifically, static analysis investigates risks from the app's source code without running the app, so it cannot ensure that valid code segments serving security and privacy compliance will be executed at run-time. Dynamic analysis overcomes the weakness of static analysis by observing the actual behavior of the app at run-time. However, dynamic cannot cover all potential risks of the app. Specifically, to observe the app's behavior, it is necessary to interact with the app's user interface (UI). However, given the complexity and diversity of the app's UI, it is completely impossible to create a general dynamic analysis workflow that can be applied to all apps [43]. This results in poor scalability of dynamic analysis, and frameworks often have to make a trade-off between accuracy and scalability [52]. In addition, dynamic analysis may demand high resources, require significant time for analysis, or be misled by apps attempting to mimic behaviors to avoid detection (e.g., pretending to ask for user consent and then disregarding user responses) [71]. Ultimately, hybrid analysis is more balanced as it leverages the advantages of both static and dynamic analysis, but also inherits the disadvantages of each approach, including poor scalability and extremely high implementation costs. In this thesis, we do not completely eliminate traditional analysis; instead, we develop methods that overcome its weaknesses.

Indeed, the first contribution of the thesis is MetaLeak (cf. Chapter 5), a system based on hybrid analysis to assess the popularity of a new threat model related to the leakage of sensitive data via EXIF metadata[11] when users upload images to cloud storage, send them via instant messaging, or share them on social networks. Compared to existing work, MetaLeak introduces a completely new attack vector that is difficult to detect and can result in severe consequences, since taking and sharing images is a common behavior in users' smartphone usage habits.[12] Specifically, we found that apps can access and leak sensitive data from EXIF metadata embedded in images without explicit user consent, due to security vulnerabilities in the Android OS itself. Particularly, this security vulnerability can be exploited as a side-channel attack through the public storage (e.g., SD card) of an Android phone. This completely violates the *lawfulness* and *transparency* criteria of GDPR. However, MetaLeak is a semi-automated framework. Specifically, we require human participation to perform the image selection and upload process through the image sharing features, if available on the app UI, to ensure accuracy. This is the weakness of the dynamic analysis we presented above that leads to MetaLeak's poor scalability.

[11] Exchangeable Image File Format (EXIF) metadata, automatically embedded in image files when they are taken, may contain sensitive information such as the exact GPS location, device identifiers, date and time, and camera settings, which can inadvertently expose users' personal data.https://www.media.mit.edu/pia/Research/deepview/exif.html

[12] https://photutorial.com/photos-statistics/

Recognizing the shortcomings of traditional analysis methods, in the second contribution, we propose the use of Large Language Models (LLM) as a potential solution to overcome these weaknesses (cf. Chapter 4). This work evaluates the prospects of LLM in supporting traditional analysis methods. Indeed, LLM are deep learning models trained on massive datasets with billions of parameters, which have demonstrated expert-level performance across a wide range of domains, including security and privacy. LLM can support static analysis through its ability to code summarize [3], detect bugs [85], reproduce bugs [21,36], and exploit vulnerabilities [20]. In addition, LLM can perform automatic interaction with the app's UI by remembering the state of the current interaction and reasoning the following steps, thus making it a perfect replacement for dynamic analysis. However, LLM are not entirely flawless due to the limitations of input token length and the need for enhanced contextual understanding to improve accuracy (cf. Section 2.3). Therefore, effective strategies are required to ensure their optimal use.

Indeed, as our third contribution, we propose ALIBIS (cf. Chapter 6), a framework based on an effective code summarization strategy using LLM to estimate the risk of leaking sensitive metadata (i.e., the threat model we identified in MetaLeak). ALIBIS is an entirely automated framework that can replace the dynamic analysis method in MetaLeak (i.e., no human intervention required), which has poor scalability. In addition, ALIBIS proposes an optimal and easy-to-integrate solution with Android apps, compared to current solutions, to mitigate the risk of sensitive metadata leakage when sharing images online.

While the first, second, and third contributions mainly concentrate on the smartphone ecosystem, the fourth contribution, WearLeak (cf. Chapter 7), shifts the focus to the wearable ecosystem. Although Wear OS is a streamlined and customized version of Android OS, the wearable ecosystem possesses several distinct and unique characteristics that are exclusive to it (e.g., companion, standalone, and embedded apps) (cf. Section 2.2). Therefore, the security and privacy risks in the wearable ecosystem are even more complex than those of smartphones, but there are fewer studies on wearables. In addition, existing studies primarily employ traditional analysis, which has numerous shortcomings, and therefore only examine certain sensitive data categories and specific data receiving points (e.g., Facebook SDK). Thus, WearLeak proposes a method based on LLM and knowledge Graphs (KG). This approach enables a comprehensive assessment of the risk of leakage of 14 categories of sensitive data, as defined by Google (see Table 2.1), and their destinations (e.g., app backend, third-party services registered/unregistered in the Manifest file).

The following subsections will present in more detail the four main contributions of this thesis.

1.1 Contribution

This thesis aims to analyze privacy compliance in accordance with the GDPR and similar data protection regulations within the smartphone and wearable device ecosystems. In addition, the thesis proposes LLM-based methods as a novel approach that supports

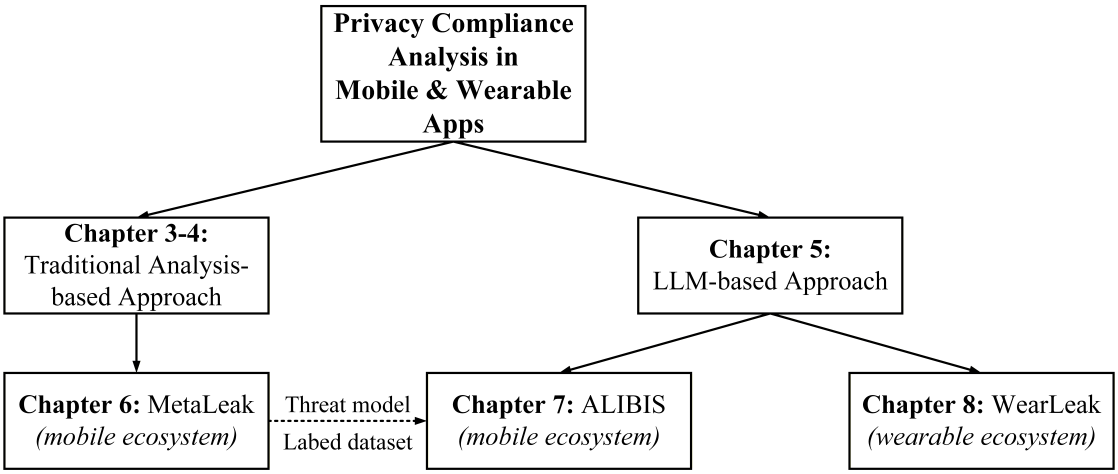

Figure 1.1: Key contributions of the thesis

and can gradually replace traditional analysis methods in evaluating privacy compliance. Figure 1.1 illustrates the relationship among the main contributions of this thesis.

Specifically, the contributions of this thesis are developed in two directions, namely, **traditional analysis-based approach** and **LLM-based approach**.

In the first direction, we employ a traditional analysis approach to assess the prevalence of a new threat model involving the leakage of sensitive data through EXIF metadata when sharing images online (Chapter 5). The results, when applying our MetaLeak framework to 5,000 popular apps, show that 21.9% ($\approx$ 1095 apps) leaked at least one of the five types of sensitive data embedded in EXIF metadata, including datetime, smartphone model, smartphone brand, serial number, and GPS (or commonly referred to as **sensitive metadata**). This result indicates that the newly discovered threat model is indeed widespread and compromising user security and privacy. However, we have to trade off between scalability and accuracy by applying semi-automatic dynamic analysis to MetaLeak.

Therefore, in the second direction, we first evaluate the capabilities of LLMs in analyzing privacy compliance and their potential to support and replace traditional analysis methods. Specifically, in Chapter 4, we analyze the state-of-the-art LLM-based solutions for tasks such as vulnerability detection, bug detection and reproduction, and malware detection.

The findings from Chapters 4 and 5 serve as the foundation for the contribution described in Chapter 6, where we develop ALIBIS, a fully automated framework based on an effective code summarization strategy. The contribution of ALIBIS is to create an early warning mechanism suitable for integration into the automated moderation systems of app stores (e.g., Google Play Store) instead of just finding evidence of violations as MetaLeak does. This capability could enable app store operators to reject apps from their stores, thereby reducing the risk of users being exposed to potentially dangerous apps. ALIBIS achieves an average accuracy, precision, recall, and F1 score of 0.8686,

0.8902, 0.881, and 0.8854, respectively, in k-fold cross-validation (k=5) on the labeled dataset from MetaLeak. In addition, ALIBIS offers a solution compatible with the Android OS to address the issue of sensitive data leakage through EXIF metadata.

Finally, in Chapter 7 (WearLeak), we evaluate GDPR non-compliance, particularly violations of the transparency principle, with respect to the types of sensitive data shared and their recipients. We develop a solution based on LLM and KG that provides unlimited observation into the categories of sensitive data (as defined by Android) and the destinations to which the data is sent. When applying WearLeak to 1,000 popular wearable apps based on the number of installations, we find that 67.5% of them violated the declared data collection and sharing practices, and 4.8% leaked data to undeclared third-party services.

1.2 Thesis Organization

The remainder of the thesis is organized as follows:

- **Chapter 2.** We introduce the fundamental concepts that are used by the mobile and wearable ecosystems, which are the target domains for this thesis. Additionally, we provide an introduction to LLM along with its context enhancement techniques.

- **Chapter 3.** We first introduce the tools and frameworks commonly used in traditional app analysis. We do not target a specific threat model. Instead, we present the general workflow of these tools and discuss their pros and cons. This is important because studies that adopt traditional analysis approaches typically rely on these tools and frameworks; therefore, this researches inherit both their advantages and their drawbacks. Next, we present state-of-the-art work using traditional analysis to analyze and evaluate the privacy compliance of mobile and wearable ecosystems. Specifically, we focus on studies directly related to the main contributions of this thesis, including: (1) privacy risks of EXIF metadata and mitigation proposals, (2) side-channel attacks leading to sensitive data leakage, and (3) privacy violations in the sharing of sensitive data and the destinations to which such data is transmitted in wearable apps.

- **Chapter 4.** We discuss recent studies that use an LLM-based approach to analyze common security risks in the mobile ecosystem (e.g., the OWASP[13] mobile top 10). Thereby, we demonstrate the potential of LLM in supporting and gradually replacing traditional methods for investigating security and privacy risks in the mobile and wearable ecosystem.

- **Chapter 5.** This chapter presents MetaLeak, a semi-automatic framework based on the traditional analysis methods (i.e., hybrid analysis) to assess the prevalence of a new threat model related to the leakage of sensitive data, including datetime,

[13] https://owasp.org/www-project-mobile-top-10/

camera model, camera brand, and GPS via EXIF metadata when users upload, share, or send their images online. Then, we assess privacy compliance with respect to sharing GPS, since this is the most easily exploitable type of sensitive information among those considered.

- **Chapter 6.** This chapter introduces ALIBIS, a fully automated framework built on a code summarization strategy that depends on how the app handles EXIF metadata. ALIBIS is considered an upgraded version of MetaLeak, designed to eliminate dynamic analysis, which suffers from poor scalability and requires human interaction with the app's UI. Finally, ALIBIS also proposes a solution for mitigating the leak of sensitive data via EXIF metadata.
- **Chapter 7.** This chapter discusses WearLeak, an automation framework that combines hybrid analysis, LLM, and knowledge graph (KG) to analyze privacy non-compliance in the wearable ecosystem.
- **Chapter 8.** This chapter presents the conclusions of the thesis and discusses directions for future research.
- **Appendix A**.This appendix provides survey results on user awareness of security risks associated with EXIF metadata.

1.3 Related Publications

The contributions of this thesis result in the following publications:

- Nguyen, Tran Thanh Lam, Barbara Carminati, and Elena Ferrari. "MetaLeak: Assessing Image Metadata Leakage in Android Apps". 2024 IEEE/ACS 21st International Conference on Computer Systems and Applications (AICCSA). IEEE, 2024 [52].
- Nguyen, Tran Thanh Lam, Barbara Carminati, and Elena Ferrari. "LLMs on Support of Privacy and Security of Mobile Apps: State of the Art and Research Directions". 2025. arXiv, https://arxiv.org/pdf/2506.11679v1 (accepted as a book chapter by Wiley-IEEE Press) [53].
- Nguyen, Tran Thanh Lam, Barbara Carminati, and Elena Ferrari. "Detecting Privacy Non-Compliance in Wearable Apps via Knowledge Graphs and LLMs". 2025 IEEE 21st International Conference on Wireless and Mobile Computing, Networking and Communication (WiMob). (accepted).
- Nguyen, Tran Thanh Lam, Barbara Carminati, and Elena Ferrari. "ALIBIS: Assessing and mitigating the risk of sensitive metadata Leakage In moBile Image Sharing". Pervasive and Mobile Computing, 2025. (under review).

Chapter 2

Background

In this chapter, we first introduce the key concepts and mechanisms shared by Android OS and Wear OS related to security and privacy, namely the permission model (PM), sandbox mechanism (SM), and data safety (DS). Next, we introduce the unique characteristics of Wear OS, such as three wearable app types (companion apps, embedded apps, and standalone apps), and the authorization mechanism specifically designed for collecting health data. Finally, we discuss LLM along with its popular context enhancement techniques.

2.1 Mobile ecosystem

The permission model (PM)[1] and sandbox mechanism (SM)[2] are Google's efforts to ensure that the Android OS and Wear OS meet the *lawfulness* criteria of the GDPR.

First, PM ensures that apps need explicit consent from users when accessing sensitive data (e.g., GPS, health data) and using sensitive hardware (e.g., camera, microphone, health sensor). PM comprises two types, namely install-time and run-time permissions. Install-time permissions are classified at a normal level, posing minimal risks to user privacy, for example, viewing network connections, preventing the phone from sleeping, running the app at startup, etc. In contrast, run-time permissions include many dangerous-level permissions that can pose significant threats to user security and privacy, for example, storage access, read health data, etc. Unlike earlier versions of Android, where apps required users to grant all permissions at the time of installation, starting from version 6.0, runtime permissions allow users to grant or deny individual permissions based on the specific features or resources they intend to use. For instance, an app has two functions, such as making phone calls and taking pictures, and requests two corresponding dangerous permissions, *READ_CONTACTS* and *CAMERA*. However, if users only utilize the phone call feature, they only need to grant the *READ_CONTACTS* permission to the app. Additionally, users can manage an app's permissions in

[1] https://developer.android.com/guide/topics/permissions/overview
[2] https://source.android.com/docs/security/app-sandbox

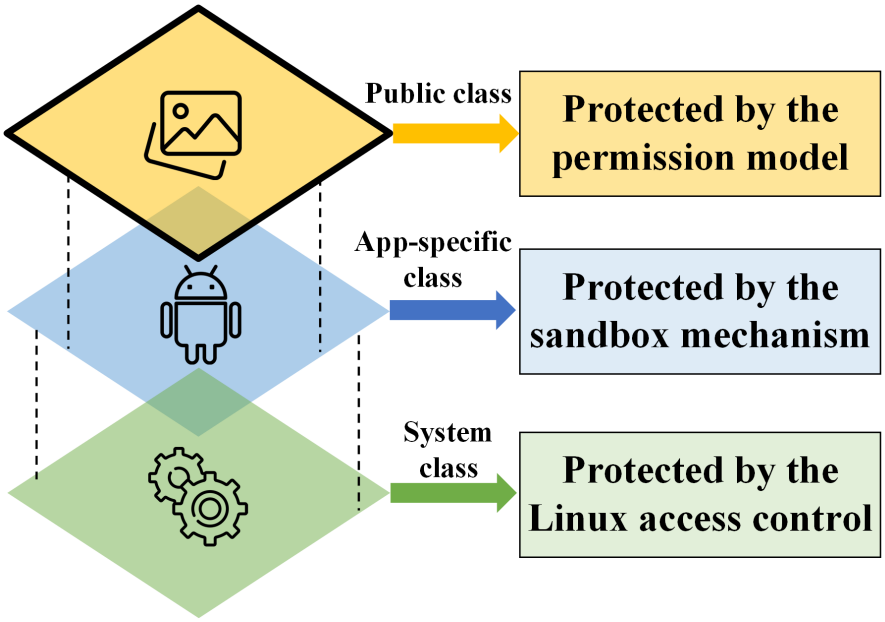

Figure 2.1: Android storage classes

the Android settings and review and modify them at any time.

To implement the PM, Google requires app developers to list all the permissions the app needs to function in the `Manifest.xml` file. This required workflow is always applied on both Wear OS and Android OS. However, Google does not check whether the number of listed permissions aligns with the app's features. This means that an app can request more permissions than necessary, mislead users into granting them, and then access sensitive data. Additionally, Google lists all permissions and the corresponding types of sensitive data protected by each permission on its official website.[3] However, this information has no value in protecting users' privacy, but simply provides information when they actively search. In practice, users, who often have a limited understanding of security and privacy, are solely responsible for granting app permissions without adequate support.

On the other hand, the SM supports PM by creating an isolation environment that ensures apps work independently through three main features. Firstly, process isolation makes sure that each app operates in its own process. Therefore, even if two apps are running on the same device, they can't directly interfere with each other's processes. Secondly, file system isolation creates independent storage folders for each app. This method prohibits other apps from accessing or modifying the app's data without permission. Thirdly, inter-process communication (IPC) strictly monitors apps' communication to prevent unwanted data exchange.

However, Android offers three data storage classes with various security levels and protection mechanisms, including (1) system, (2) application-specific, and (3) public [8],

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

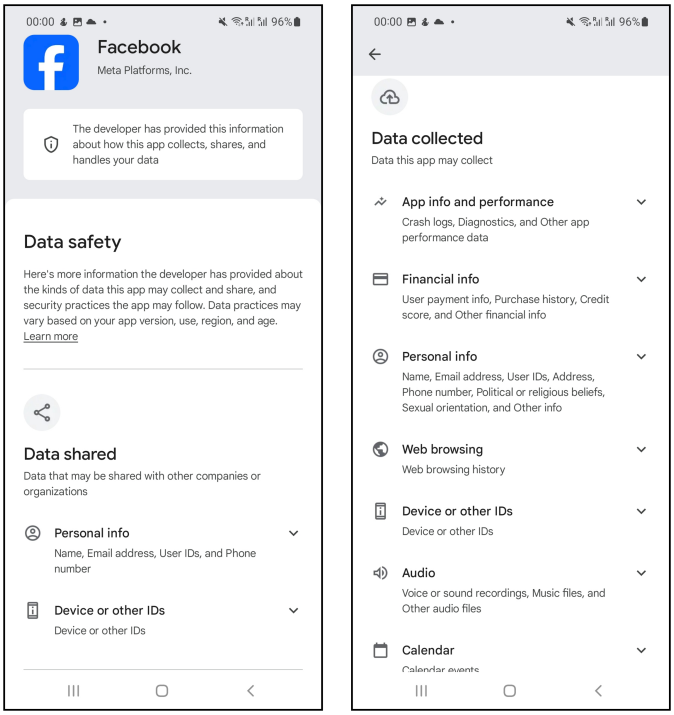

Figure 2.2: Data Safety of Facebook apps[4]

as illustrated by Figure 2.1. The system storage class contains the whole Android OS and is secured by Linux access control. In contrast, the application-specific storage class is controlled by a specific app and can only be read and written by this app, which is commonly mounted at the path /data/. Application-specific storage is the class protected by the SM. Finally, the public storage class, which is often mounted at the path */sdcard/* or */mnt/sdcard/*, is used to share data among apps, primarily storing multimedia files created with the camera and microphone. Additionally, it is accessible from a computer connected via USB to the smartphone. Public storage is only protected by PM, meaning that any app with read (*READ_EXTERNAL_STORAGE*) and write storage (*WRITE_EXTERNAL_STORAGE*) permissions can access this storage area [94].

As of July 20, 2022, Google requires app developers to declare all types of sensitive data that their app will collect and share in the Data Safety section of the Google Play console. For example, the Data Safety section of the Facebook app is shown in Figure 2.2. This is Google's measure to ensure *transparency* criteria according to GDPR regulations. Data Safety helps users understand what types of private information the app may collect and share, allowing them to decide whether to install it. However, this information doesn't show itself (e.g., as a pop-up or notification) when the user installs the app. Instead, users must proactively check the information in the Google Play

[4] https://play.google.com/store/apps/datasafety?id=com.facebook.katana&hl=en

console. As a result, the Data Safety section does not adequately secure users' privacy. Finally, the Data Safety section only indicates the category of sensitive data without specifying which particular data types fall under that category. Therefore, to create a list of sensitive data types, we first refer to Google's official documentation on permissions.[5] 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.[6]. Finally, we obtain Table 2.1, which summarizes the 14 sensitive categories along with their corresponding data types.

Table 2.1: Sensitive data categories and corresponding data types.

No	Sensitive category	Data type
1	Device or other IDs	IMEI, MAC address, Widevine Device ID, Firebase Installation ID, Advertising ID (AD ID), IP address
2	Personal info	Full name, username, password, email address, user ID, phone number, birthday, gender
3	Location	Longitude, latitude, altitude
4	Health & fitness	Weight, height, medical records (heart rate, etc.), exercise (step, swim, etc.)
5	App info & performance	Crash logs, app logs, CPU/RAM/battery
6	Messages	Email/SMS/MMS/chat (subject line, sender, recipients)
7	Financial info	Accounts/credit card number, transaction history
8	Photos or video	Photos and videos
9	Audio files	Audio files
10	Files and docs	Voice, sound recordings, user's music
11	Calendar	Event title, event description, event start/end time, event status, event organizer, reminder, alarm, calendar account information
12	Contacts	Contact name, contact number, call log, phone state
13	App activity	App interactions, in-app search history, installed apps
14	Web browsing	Cache or cookies

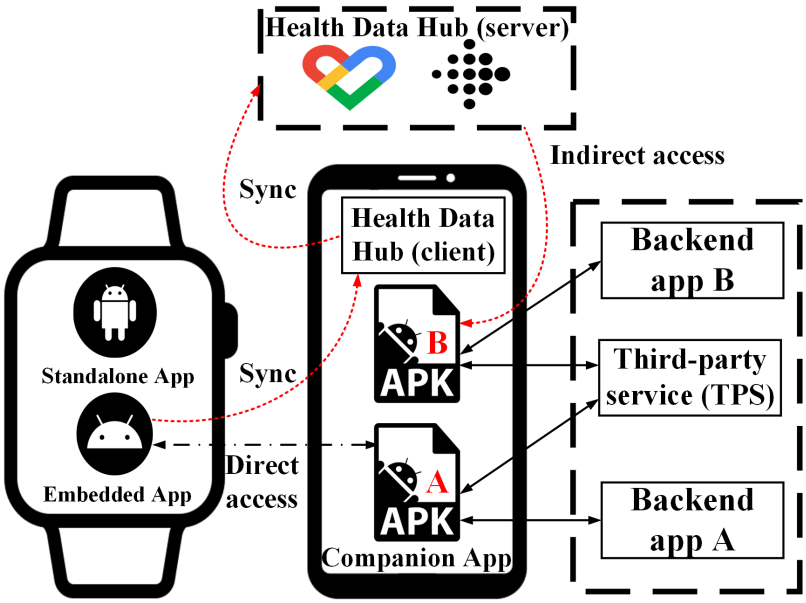

Figure 2.3: Wearable apps: Direct and Indirect health data access

2.2 Wearable ecosystem

The wearable ecosystem inherits core mechanisms, such as the permission model, sandbox mechanism, and data safety mechanisms, but it also possesses distinct characteristics unique to its architecture.

Firstly, wearable apps are categorized into three types, namely, embedded, companion, and standalone [57]. Specifically, embedded apps (for example, a heart rate monitor) run on wearable devices and are often integrated into the firmware. These apps cannot function properly or provide full functionality without the support of a companion app (for example, a fitness app) installed on a smartphone. Specifically, companion apps synchronize health data from embedded apps via Bluetooth, then perform analysis and statistics, and provide personalized exercise recommendations based on the user's health profile. Because they operate on smartphones, companion apps can share sensitive user data via the Internet, similar to non-wearable apps. In contrast, standalone apps (for example, calendar apps) can operate independently on wearable devices without connecting to smartphones. Standalone apps can also exchange data over the Internet, but this is limited because it requires the wearable device to be equipped with a Wi-Fi/LTE chip, which increases the device's cost, size, and battery consumption. In addition, Google recommends minimizing data exchange in standalone apps.

Secondly, Wear OS introduced two methods for the companion app to access health data, namely direct and indirect access, as illustrated in Figure 2.3. Specifically, direct

[5] https://developer.android.com/reference/android/Manifest.permission
[6] https://developer.android.com/privacy-and-security/declare-data-use

health data collection uses a permission model. That is, users explicitly consent to the companion apps collecting data directly from the embedded app. This is done by granting the permissions to read health sensors (e.g., *READ_HEART_RATE*[7]), as shown in the example of the companion app A in Figure 2.3. However, health data is extremely sensitive information, so device manufacturers often limit companion apps from receiving data directly from wearable device sensors. For example, Samsung only allows 15 companion partners to directly access health data on their device (Galaxy Watch)[8]. Instead, the indirect health data reading mechanism via the Health Data Hub[9] (e.g., Google Fit[10], Google Health Connect, Fitbit, Samsung Health) is used as an alternative, as shown in the example of the companion app B in Figure 2.3.

Particularly, health data is synchronized from the embedded app to the Health Data Hub (client) running on the smartphone. Then, the Health Data Hub (client) will send health data to the Health Data Hub (server), which is hosted on the cloud infrastructure of the wearable device manufacturer. When the companion app B wants to access health data, users authorize the app via the OAuth2[11] protocol by logging into their Google accounts through a pop-up interface provided by this app. Upon successful login, Google issues an authorization code to the app B. After that, the app B sends the authorization code to Google's authentication server to obtain an access token. Finally, app B uses this access token to retrieve health data indirectly from the Health Data Hub (server).

2.3 Large Language Models (LLM)

This section presents the fundamental concepts of large language models (LLM) [48, 67] and their applications, with a primary focus on security and privacy applications. Furthermore, we examine two important approaches for enhancing the contextual awareness of LLM, namely, fine-tuning and prompt-engineering, and then explain why we choose prompt-engineering in this thesis. Finally, we introduce three prompt-engineering techniques used in this thesis, that is, Few-shot Learning (FSL), Retrieval-Augmented Generation (RAG), and Graph-based Retrieval Augmented Generation (GraphRAG).

2.3.1 Fundamental concepts of LLM

Artificial intelligence (AI) is not a novel concept because the first ideas of AI were introduced in the 1940s-1950s, while Eliza Chatbot, the first functioning Generative AI (GenAI), was launched in the 1960s-1970s [91].

[7] https://developer.android.com/reference/android/health/connect/HealthPermissions#READ_HEART_RATE

[8] https://www.samsung.com/ca/apps/samsung-health/

[9] These apps are provided by wearable device manufacturers for managing and monitoring the wearable devices.

[10] Google Fit API will be deprecated after 2026. New registrations are no longer accepted, but existing accounts can continue to use the API until 2026. https://developers.google.com/fit/rest

[11] https://oauth.net/2/

GenAI [93] is a subset of AI that excels in creating new content such as articles, poems, music, paintings, and films. LLM [48] is a subset of GenAI that primarily focuses on understanding human natural language and generating language-related material, including translation, text summarization, and programming. LLM can comprehend language's statistical and semantic characteristics using text-based training datasets.

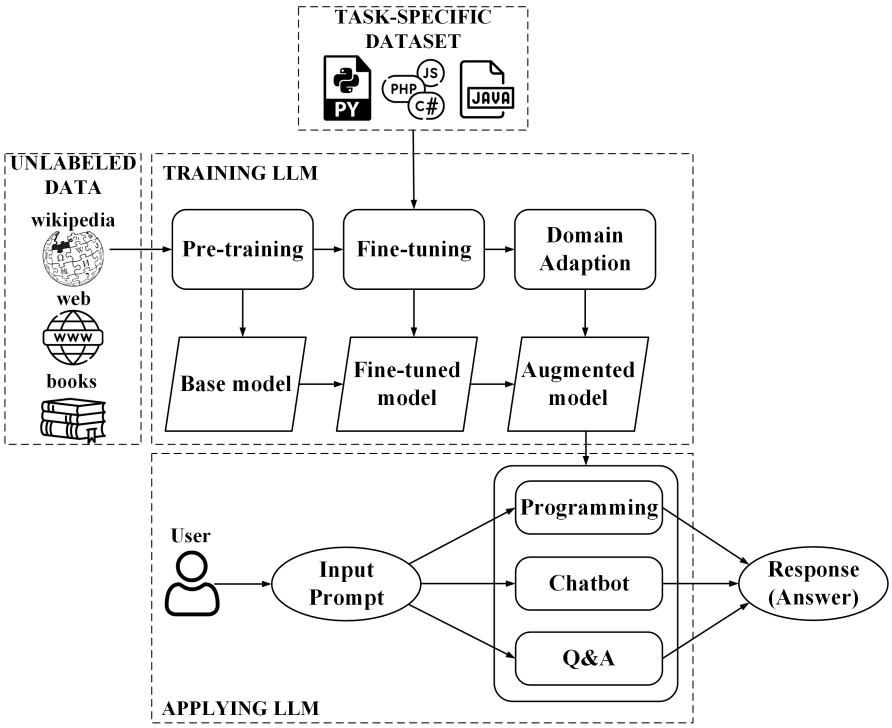

Figure 2.4: LLM overview

Figure 2.4 gives an overview of the LLM reference architecture. The LLM training process consists of three main steps, namely, pre-training, fine-tuning, and domain adaptation [50]. First, the pre-training step receives a large amount of unlabeled text data (e.g., Wikipedia, books, and website data). For example, the Generative Pre-trained Transformer 3 (GPT-3) model is trained from a common crawl dataset (web pages), the BookCorpus dataset (11,000 books), Reddit articles, and Wikipedia [29]. Pre-training uses unsupervised learning (without human intervention) to build a base LLM. For instance, OpenAI requires the base model (GPT) [2] to predict the next word in an incomplete sentence, whereas Google trains BERT [13] using the Masked Language Modeling method; precisely, the model must predict masked words in a sentence. The base model can generally understand natural language (e.g., grammar, syntax, and semantics) but is not specialized for any specific task. Thus, the fine-tuning step aims to obtain a fine-tuned model adaptable to a specific scenario. In the fine-tuning step, the base model is trained using supervised learning on a smaller and task-specific dataset. The base model is provided with inputs with corresponding outputs labeled by humans. For example,

Code Llama [64] is a fine-tuned model for programming-related tasks built on top of the base model Llama [79]. Finally, in the domain adaptation step, experts in a particular domain adjust the fine-tuned model to obtain an augmented model for specific real-world applications, such as chatbots, question-and-answer (Q&A), and programming. For example, ChatGPT[12] is designed explicitly for chatbots, based on variations of the GPT model.

To use LLM, the user needs to build a *prompt* as input. The prompt is a string of characters, a paragraph, or a question that the user provides to an LLM to perform a specific task. In addition, the prompt may include hints and specify LLM's role to enhance the model's reasoning ability. For instance, we can formulate the prompt to instruct the LLM to assume the role of an Android expert and develop a quick sort algorithm utilizing the Android programming language as follows: *"You are an expert in Android programming language. Please help me implement a quick sort algorithm in Android".*

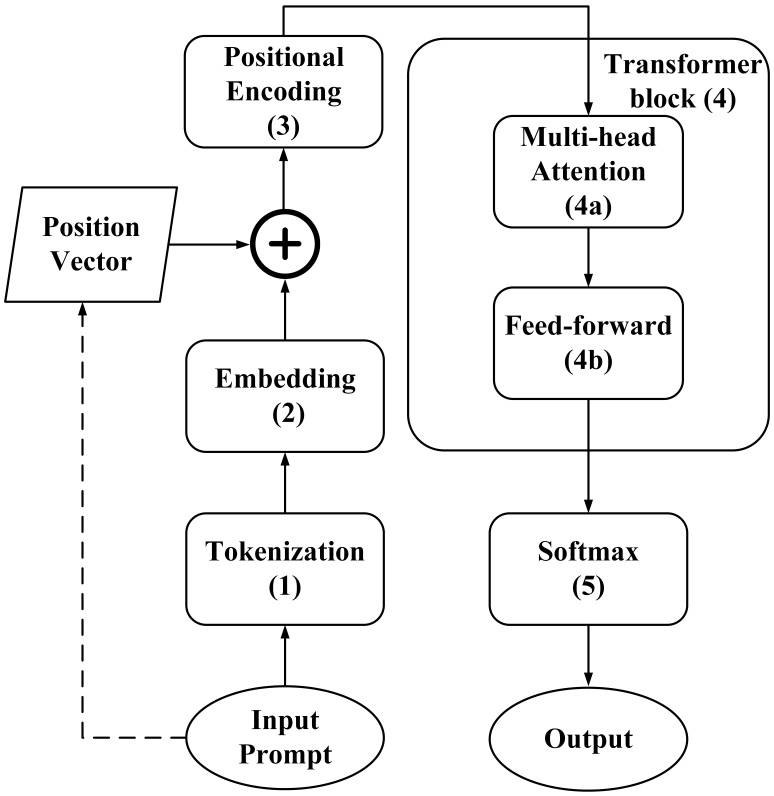

Figure 2.5: Transformer architecture [81]

It is easy to realize that training the base model is the foundation for effectively deploying LLM, and it is the most expensive process. For instance, the training cost of

[12] https://openai.com/index/chatgpt/

GPT-3 is over 1 million USD, whereas for GPT-4 is over 100 million USD.[13]

Most of the current LLM, such as OpenAI's GPT family (GPT-3, GPT-3.5, GPT-4) and Google's Gemini [78], are based on the transformer architecture (see Figure 2.5) for training the base model [58], rather than employing older architectures like RNN (Recurrent Neural Network) and its variants [68]. Therefore, the following discussion focuses on the transformer architecture.

The transformer model can perform many language-related tasks, including document translation, paragraph composition, poetry creation, and language translations. Nonetheless, the process and objective of a transformer model are the same regardless of the specific tasks for which it is employed. Specifically, a transformer relies on input prompts to predict the words that should be used to complete a sentence, paragraph, or poem. Specifically, the transformer receives a prompt as input and calculates the probability of the words that can be used as the next word to complete the sentence in the output. Next, the transformer will select the word with the highest probability to fill in the incomplete sentence. For example, suppose the input prompt is an incomplete sentence: *"I go to school, and ..."*. The goal of the transformer is to calculate the probability of the following word to fill the *"..."* in the incomplete sentence. Specifically, the transformer chooses a collection of possible words (e.g., *"friend", "teacher", "mother", "father", "my", "her", "his"*, etc.) from the unlabeled data in Figure 2.4 and then calculates their probabilities of being the good candidates to fill the incomplete sentence. Next, the transformer chooses the word with the highest probability as the following word for the sentence, for example, *"my"*. After that, the transformer has a new input that is an incomplete sentence *"I go to school, and my ..."*. The transformer repeats finding the word with the highest probability as the subsequent word to fill the incomplete sentence until it returns a final output (i.e., a complete sentence), for example, *"I go to school, and my friends go to the cinema"*.

To illustrate the working of the transformer architecture, suppose the input prompt to be the incomplete sentence *"I go to school, and ..."*. The following is a detailed description of how the transformer produces an output that is a complete sentence.

First, the input prompt goes through the tokenization process (1) (see Figure 2.5) to divide the text string into smaller parts called tokens, in which each token represents a word or a character. For example, the input prompt is divided into 6 tokens, namely *"I"*, *"go"*, *"to"*, *"school"*, *","* and *"and"*.

Subsequently, tokens are transformed into a numeric vector (aka embedding vector) (2) to facilitate the model in the computation required in the following steps.

After that, the positional encoding step (3) incorporates a distinct *position vector* into each embedding vector by vector addition, forming a *position-encoded embedding vector*. Position vectors are calculated based on the word's position in the input prompt and encode the absolute position of a token within a sentence. The position vector is essential because the transformer does not make sequential observations of each word in the sentence, like RNNs, but only focuses on a few important words. Thus, changing the position of words in the sentence can cause the model to predict incorrectly. The

[13] https://www.statista.com/chart/33114/estimated-cost-of-training-selected-ai-models/

positional encoding mechanism guarantees a distinct vector for each word in the input prompt. This leads to a distinct aggregation vector for the whole sentence. As a result, different vectors will represent sentences made up of the same words but in a different order. For instance, suppose we have the following sentences: *"I go to school, and my friends go to the cinema"* and *"I go to the cinema, and my friends go to school"*. Although made up of the same characters but with a distinct arrangement, the two sentences provide two different contexts. Formulas for calculating position vectors are based on the sine and cosine functions. We refer the interested readers to [81] for more details.

Next, the transformer block (4) receives position-encoded embedding vectors as input. Its output is a list of words that can be used to fill in the incomplete position *("...")* in the input prompt. Each transformer block has two main components: multi-head attention (4a) and feed-forward (4b).

Because multi-head attention is built on the self-attention mechanism, we explain the self-attention mechanism first. The self-attention mechanism, instead of considering the whole sentence, only focuses on the words that are most relevant to the considered context. Human natural language is extremely complex, especially with many conjunctions, prepositions, punctuation, etc., to link ideas together. However, not all words in a sentence contain important information. Thanks to the self-attention mechanism, the transformer reduces the amount of needed computation. Specifically, the self-attention mechanism calculates the attention score to evaluate the relevance of each element in position-encoded embedding vectors. This is equivalent to rank the relevance of each word (*"I"*, *"go"*, *"to"*, *"school"*, *","*, and *"and"*) in the input prompt.

The attention score represents the relevance of a specific word to the other words in the input prompt, and a higher value implies that the word carries more context than the others, making it more important, so the transformer will focus on this word. The attention scores are aggregated into a context vector (i.e., the output of self-attention mechanism) that allows the model to recognize what word's position needs attention and, from there, predict the following output [55]. For example, in the input prompt, the words *"I", "go" and "school"* are more important than *"to", "and", and ","*. Therefore, for example, the context vector of the input prompt would be as follows: *["I", "go", "to", "school", ",", "and"]* $\rightarrow$ context vector = [0.7, 0.9, 0.2, 0.9, 0.1, 0.2] with higher attention score values at the word positions that require focus.

In practical architectures, transformers utilize multi-head attention, an upgrade of self-attention, to achieve parallel computing. The *"heads"* in multi-head attention execute attention score calculations numerous times concurrently rather than just one time, as in self-attention. Each *"head"* calculates attention scores based on many aspects of the sentence, including syntax, semantics, and word relationships. For instance, in the input prompt (*"I go to school, and ..."*), *head-1* focuses on the words *"I"* and *"go"* to capture the sentence's grammatical structure. Meanwhile, *head-2* concentrates on the word that provides context, such as *"school"*. The outcomes from each head are combined to form a final context vector (final output) and carry on a more comprehensive semantic representation.

Next, feed-forward (4b) is a multi-layer neural network that applies two linear transformations with nonlinear activation functions, such as ReLU. Feed-forward receives the context vector from multi-head attention as input and then reshapes it to help the transformer learn the contextual features of selected attention words in the input prompt. The feed-forward output is a prediction list of words that might fill in the incomplete position in the input prompt. For instance, in our running example, feed-forward learns the contextual features of required attention words *"I", "go" and "school"* to determine that we need possessive adjectives, such as *"my", "her", "his"*, etc., as the next word in the sentence.

Finally, softmax (5) computes the probability for the words selected by the feed-forward step and chooses the word with the highest probability as the following word in the sentence; for example, *"my"* in our running example. All the steps from (1) to (5) are then repeated until the sentence is complete, for example, *"I go to school, and my friends go to the cinema"*.

Table 2.2: LLM for code generation and analysis.

Model name & Developer	**Supported Programming Languages**	**Applications**
CodeBERT (Microsoft) [22]	Python, Java, JavaScript, PHP, Ruby, Go	Code search & document generation
GraphCodeBERT (Microsoft) [28]	Python, Java, JavaScript, PHP, Ruby, Go	Code search, Code clone detection, Code refinement, Code translation
GPT-3/GPT-3.5/ GPT-4 & variants (e.g., GPT-3.5-turbo, GPT-4 turbo, GPT-4o) (OpenAI) [2]	Python, JavaScript, Go, Perl, PHP, Ruby, Swift, TypeScript, Shell	Code generation, Code summarization, Code refactoring, Code executing
Code Llama (Meta AI) [64]	Python, C++, Java, PHP, TypeScript, C#, Bash	Code generation, Code infilling

In addition to answering questions, translating, and summarizing pure text, LLM demonstrate excellent abilities in supporting coding activities. With this capability, LLM can be an effective solution to replace traditional analysis methods in analyzing app security and privacy violations (cf. Chapter 4). More precisely, LLM have many applications in the programming field, including (1) *Code search & document generation*: LLM can search and understand the semantic relationship between programming languages and natural languages to represent them in documents, supporting programmers in reading and understanding code; (2) *Code clone detection*, that is, detecting code segments that produce similar results with the same input; reducing this duplication helps reduce software maintenance costs and prevent bugs; (3) *Code refinement*, that is, automatically fixing bugs; (4) *Code translation* to support migrating old software from current programming languages to another one; (5) *Code generation*, that

is, the ability to automatically write code from natural language descriptions; (6) *Code summarization*, that is, the ability to explain the meaning, purpose or logic of code; (7) *Code refactoring*, that is, the ability to improve the structure of code to make the code more concise and optimized; (8) *Code executing*, that is, the ability to run code and obtain results without human involvement (for example, installing the development environment and compiling code); (9) *Code infilling*, that is, the ability to complete the missing parts of the code based on the context surrounding the missing position. Table 2.2 lists representative LLMs for code generation and analysis. For each of them, the Table 2.2 reports the developers of the model, the programming language it supports, and the major code-related applications it facilitates.

According to Table 2.2, we can select the suitable LLM for specific tasks. For example, Code Llama supports Java and code generation, which is more suited for supporting programmers in developing mobile apps. In contrast, the GPT model and its variants have code summarization ability, which is more appropriate for understanding code logic and detecting security vulnerabilities.

Although LLM can understand both natural language and programming languages, it still has two major disadvantages. Firstly, LLM are non-task-oriented models [14] because they were not trained for any specific task. Specifically, as presented above, LLM primarily utilizes the self-attention mechanism to predict the next word based on probability calculations, making it susceptible to hallucinations if the input prompt lacks a specific context. This reason leads to LLM often giving vague responses. This reduces their effectiveness, especially for tasks involving the analysis of security and privacy violations, which require precise proof and explanations.

Secondly, LLM cannot handle input prompts of infinite length. Instead, LLM are limited by the numer of tokens input[14]; for example, GPT-4o[15] only supports 128,000 tokens. This limits LLM's ability, especially for code summarization, because they cannot process the entire source code of the app. In summary, to use LLM effectively, we need to apply context-enhancement techniques within the constraints of input token limits. In the next section, we discuss popular context enhancement mechanisms for LLM.

2.3.2 Fine-tuning & Prompt-engineering

Fine-tuning and prompt engineering are two popular techniques to enhance context and thereby improve LLM accuracy.

Fine-tuning [69] is a technique of training a base model using a small amount of labeled data to obtain a fine-tuned model that is more efficient and accurate for a specific task. In short, the objective of fine-tuning is to retrain some of the base model's parameters, thereby expanding the model's context with the extra data.

In contrast, prompt engineering [69] is a completely different approach. The goal of prompt engineering is not to retrain the base model, but rather to utilize different techniques (e.g., few-shot learning) to construct a more context-rich input prompt, thereby

[14]Token is a unit of measure for the amount of text that LLM can process in a single input prompt.
[15]https://platform.openai.com/docs/models/gpt-4o

enhancing the accuracy of the base model.

Both fine-tuning and prompt engineering have their own advantages and disadvantages, as summarized in Table 2.3.

Specifically, fine-tuning requires a lot of hardware and time to retrain the model. In addition, fine-tuning requires in-depth knowledge of the model to prepare labeled data that is compatible with the format used in the base model (for example, the GPT-4o model requires labeled data to be in JSONL format[16]). Finally, fine-tuned models lack flexibility because they must be retrained whenever new training data is added. However, fine-tuning does not require complex prompts. That is, given the same input prompt, the fine-tuning model will give a more accurate response than the base model. This helps the fine-tuning avoid the token limitation of the input prompt.

Meanwhile, prompt engineering does not require high hardware costs and has a fast deployment speed. Additionally, this technique is highly flexible in updating the context within the input prompt and does not require any specific format. However, since the contextual information is directly embedded in the prompt, this approach may exceed the LLM's token limit if the context is too long.

Finally, in terms of accuracy, prompt engineering gives higher accuracy than fine-tuning in code summarization [82].

Therefore, in this thesis, we choose **prompt engineering** to enhance the context and accuracy of LLM.

Table 2.3: Comparison of Fine-tuning and Prompt Engineering Techniques

Aspect	Fine-tuning	Prompt Engineering
Hardware Cost & Training Time	High hardware cost and considerable time for retraining.	Low hardware cost and fast deployment.
Expertise Requirement	Requires in-depth knowledge to format labeled data compatible with the base model	Does not require specific data formatting or deep model knowledge.
Flexibility	Low flexibility—must retrain the model to update new training data.	High flexibility—context can be updated directly in the input prompt.
Prompt Complexity	Simple prompts avoid the token limit	Complex prompts to provide adequate context may exceed token limits
Accuracy (Code Summarization)	Lower	Higher

In the next section, we discuss the concepts, workflow, and advantages/disadvantages of three prompt-engineering techniques applied in this thesis: Few-shot Learning

[16] https://platform.openai.com/docs/guides/model-optimization

(FSL), Retrieval-Augmented Generation (RAG), and Graph-based Retrieval Augmented Generation (GraphRAG).

2.3.3 FSL & RAG & GraphRAG

Few-shot Learning (FSL) and Retrieval-Augmented Generation (RAG) are two popular prompt engineering techniques, while Graph-based Retrieval Augmented Generation (GraphRAG) is a variant of RAG that combines RAG with a Knowledge Graph (KG).

2.3.3.1 Few-shot learning (FSL)

FSL [72] is a machine learning technique inspired by how humans learn and think. Specifically, humans can reason and make decisions informed by past experiences, even when the present issue differs from prior lessons. Therefore, the goal of FSL is to achieve the highest performance (P) with the least number of labeled examples provided (E) for a specific task (T) [84].

More precisely, the FSL final prompt consists of two components, namely an example collection and a user's prompt, as shown in Figure 2.6. The example collection consists of labeled $\langle \text{input}, \text{output} \rangle$ pairs related to the user's prompt. The $\langle \text{input}, \text{output} \rangle$ pairs help enhance the context of the LLM, thus increasing the model's accuracy.

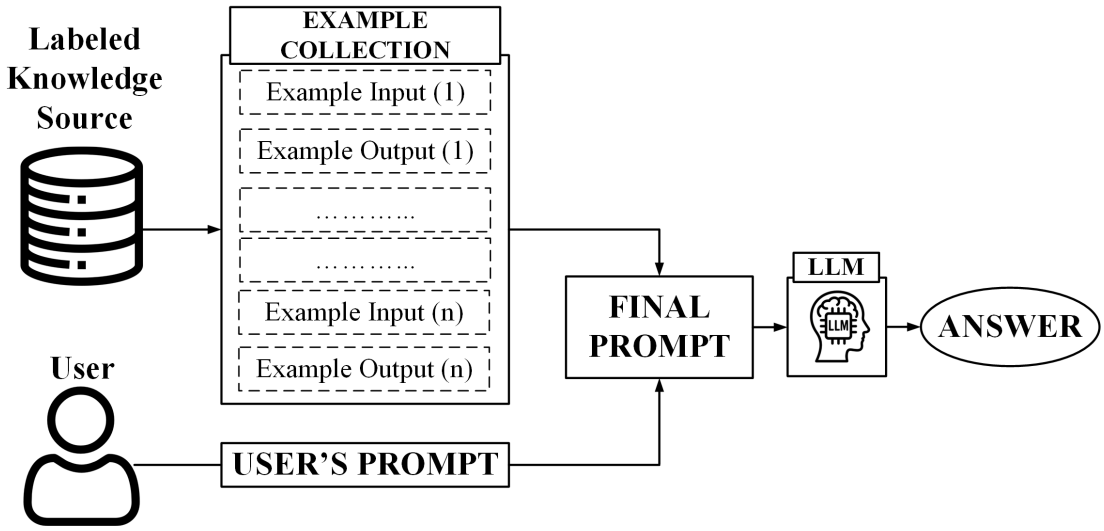

Figure 2.6: FSL overall architecture

FSL learning is suitable for classification and recognition problems when collecting all training samples is impossible [72], such as classifying scam emails. For instance, suppose we have a user's prompt as follows: *''Dear Customer, Congratulations! You are among 10 lucky winners of a free USA tour. Please verify by clicking the link below and then receive the flight details"*. The example collection and user's prompt will be combined into a final prompt: *"You are expert security. Kindly assess if the subsequent email* $\{user's\ prompt\}$

Table 2.4: FSL example collection for scam mail classification.

Example collection	**Input**	**Output**
Example-1	Dear customer, You have a gift from our company. Please click the link below and fill out your information.	Scam
Example-2	Dear Customer, Banking services will be temporarily unavailable from 12 AM to 5 AM for maintenance. No action is required. Thank you.	Not Scam
Example-3	Congratulations! You have won a $5,000 gift card. Click here to claim your prize.	Scam

is a scam or a legal communication. Utilize the information from the following examples: $\{example-1\}$, $\{example-2\}$, *and* $\{example-3\}$ *".* The $\{example-1\}$, $\{example-2\}$, and $\{example-3\}$ are variables used to pass the corresponding $\langle \text{input}, \text{output} \rangle$ pairs listed in Table 2.4.

However, FSL has two main disadvantages. First, we cannot provide too many examples in the FSL prompt since the limitation of LLM token input, as discussed above. Second, the examples provided to the LLM may not be optimal for the input prompt. That means there is no mechanism to evaluate the relevance of the examples to the input prompt, and the examples are selected based on the prompt designer's experience. As a result, more appropriate examples that better match the input prompt may be overlooked.

2.3.3.2 Retrieval-Augmented Generation (RAG)

RAG (aka base RAG) [66] overcomes FSL's weaknesses in selecting the example collection for context enhancement by using a mechanism to assess the relevance of the examples with the input prompt instead of relying solely on the user's experience.

The RAG architecture has three main stages, namely (1) preparation, (2) retrieval, and (3) generation, as shown in Figure 2.7. In the preparation stage, labeled data (knowledge source) is transformed into embedding vectors via embedding models (e.g., OpenAI's text-embedding-ada-002[17]). The objective of the embedding model is to represent raw texts (such as words, phrases, or code) in a multidimensional numeric vector space so that semantically similar texts have the smallest distance in the space. Next, embedding vectors are stored in a vector database (Vector DB) (e.g., Faiss[18] or ChromaDB[19]).

[17] https://platform.openai.com/docs/guides/embeddings/embedding-models
[18] https://github.com/facebookresearch/faiss
[19] https://www.trychroma.com/

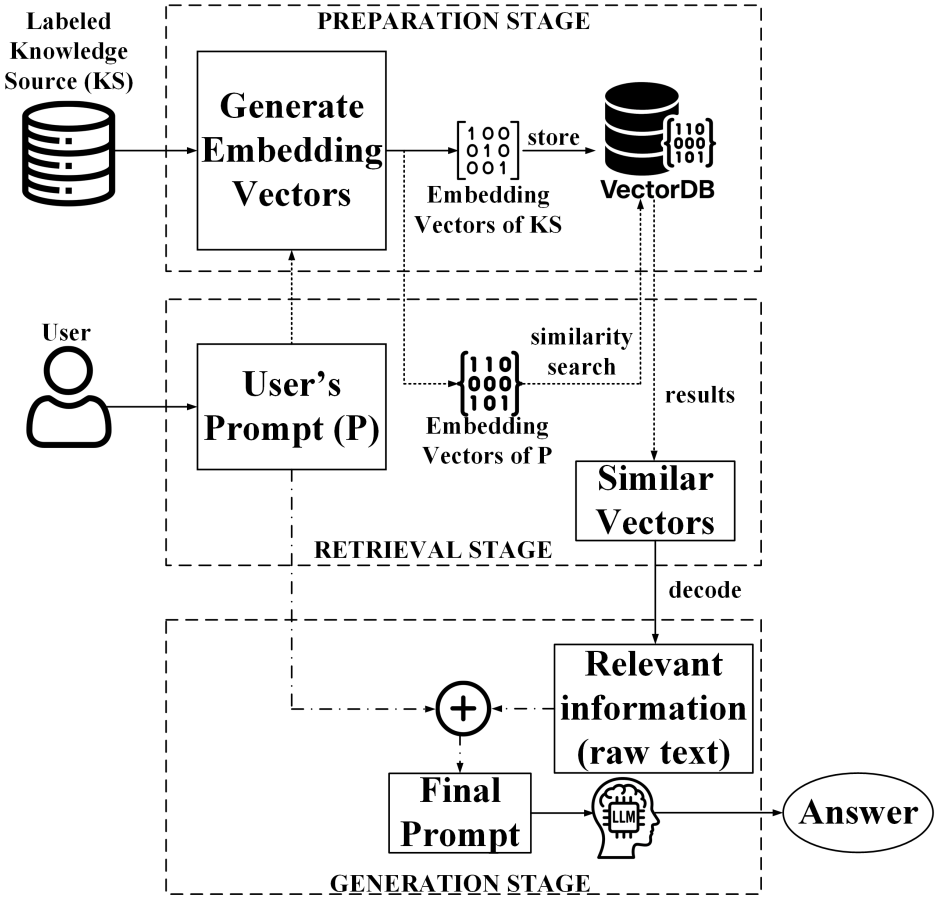

Figure 2.7: RAG overall architecture

In the retrieval stage, suppose a user sends his/her prompt (P) to LLM. The user's prompt will first be converted into embedding vectors (v_p) using the same embedding models used in the preparation stage. Next, v_p is used to query the vectorDB to find similar vectors through a similarity search. Similar search uses calculations such as cosine similarity to determine the similarity between embedding vectors.

Finally, in the generation phase, similar vectors are decoded into raw text to form a set of relevant information that enhances the context for the LLM. Next, the user's prompt (as raw text) is combined with relevant information to form the final prompt for the LLM. Basically, a prompt in RAG is modeled as follows:

$$relevant\_information \leftarrow retrieve(user\_prompt) \quad (1)$$

$$final\_prompt \leftarrow relevant\_information \parallel user\_prompt \quad (2)$$

Where $user\_prompt$ is the prompt entered by the user, and $relevant\_information$ is a set of relevant information retrieved from the vector DB based on the $user\_prompt$.

Table 2.5: RAG for code summarization.

Data	**Code block (Raw text)**	**Embedding vector**	**Label of Code Block**
1	def add(a, b): return a + b	[0.015708842089961785, . . . , -0.0258686572771857]	Sum function
2	def subtract(a, b): return a - b	[0.004827093476324077, . . . , -0.012001586502984194]	Subtraction function
3	def multiply(a, b): return a * b	[0.014412204954896129, . . . , -0.02305418474387719]	Multiplication function
4	def divide(a, b): if b == 0: return “Error" else: return a / b	[-0.008386097271214806, . . . , -0.017810343585481985]	Division function
User's prompt	**Code block (Raw text)**	**Embedding vector**	**Code Summarization**

5	def add(number_array):
	total = 0
	for num in number_array:
	total += num
	return total
		[0.023987490179583706,
		. . . ,
		-0.05613979951752224]
			Sum function

Table 2.5 shows an example of applying RAG to code summarization. In this particular examples, labeled data (rows 1 to 4) are the code blocks listed in the 2nd column, labeled as sum function, subtraction function, multiplication function, and division function (4th column), respectively. Then, these raw text code blocks are converted into corresponding embedding vectors (see the 3rd column). These embedding vectors are stored in the vector DB, as described in the RAG workflow. Similarly, the user's prompt (i.e., the code block listed in the 2nd column of the 5th row), is converted into embedding vectors (3rd column) with the same embedding model used for labeled data.

Next, RAG uses the embedding vector of the user's prompt to query the vector DB to retrieve similar vectors. In this case, RAG selects the embedding vector in the first row (sum function) as the most similar vector. Next, RAG converts the retrieved vector into raw text (i.e., relevant information) and combines it with the user's prompt (in raw text) to form the final prompt. For example, the final prompt is: *“You are an expert in programming. Please summarize the meaning of the function* $\{user's\ prompt\}$ *with the following contextual information* $\{relevant\ information\}$*".* At this point, the LLM rely on the user's prompt and the relevant information, which is the code block labeled as the sum function, to summarize the user's prompt. The code summarization result of the user's prompt is the sum function.

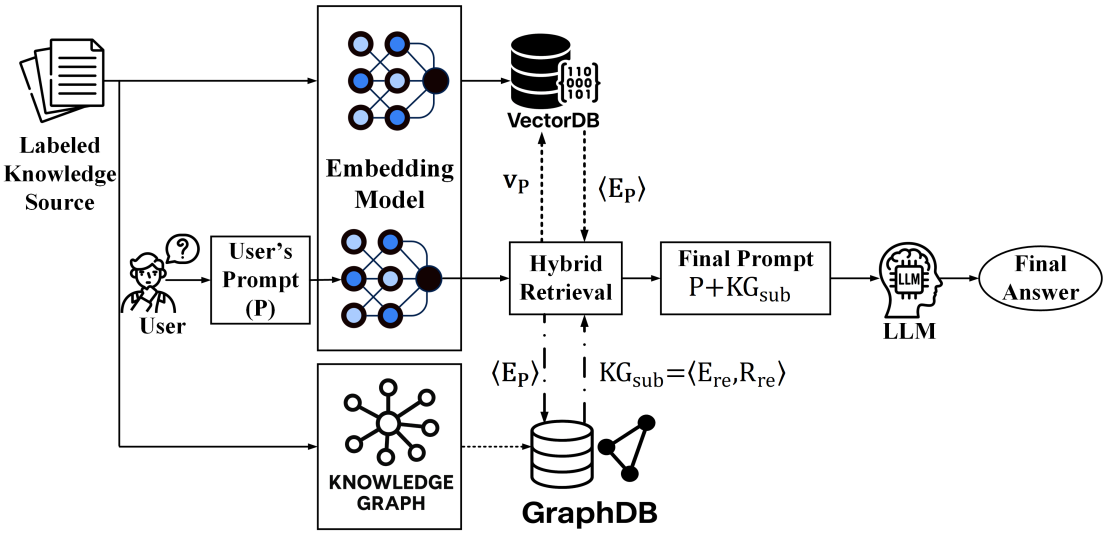

Figure 2.8: GraphRAG overall architecture

Table 2.6: Comparison of FSL, RAG, and GraphRAG

	Advantage	Disadvantage
FSL	• Easy to implement • Simple prompt design • Low cost	• Affected by token limitation • No mechanism for checking the relevance of examples
RAG	• Support checking the relevance of examples • Less affected by token limitation	• Complex deployment • High cost • Ineffective for structured and relational data
GraphRAG	• Less affected by token limitation • Effective for structured and relational data	• Very complex deployment • Very high cost

2.3.3.3 Graph-based Retrieval Augmented Generation (GraphRAG)

GraphRAG [31] is a method that combines a knowledge graph (KG) and retrieval-augmented generation (RAG) to improve the accuracy of base RAG. Specifically, base RAG has proven effective in helping LLM improve inference on problems that have never been trained (i.e., external data), for example, code summarization. Base RAG considers semantic similarities based on embedding vectors. However, data represented in embedding vector form is often discrete and unstructured, so it cannot represent the relations between data points, which can lead to a lack of context for LLM [30]. For instance, base RAG struggles with the Android app's Manifest.xml file as the external knowledge source because it is structured data with many XML tags nested following a parent-child relationship.

GraphRAG is a 3-step workflow as shown in Figure 2.8. In the preparation step, external data (documents, manifest files, etc.) is analyzed to extract entities and their relations. A knowledge graph (KG) is constructed from the extracted information and stored in GraphDB. In addition, entities are also converted into embedding vectors and stored in vectorDB (similar to the base RAG). In the hybrid retrieval step, the user_-prompt (P) is converted into an embedding vector (v_p). Then, v_p is used to query vectorDB to identify entities related to P, called $\langle E_P \rangle$. Next, based on $\langle E_P \rangle$, query GraphDB to identify all neighbor entities directly or indirectly related to $\langle E_P \rangle$ along with corresponding relations, called $KG_{\text{sub}} = \langle E_{\text{retrieval}}, R_{\text{retrieval}} \rangle$. Finally, in the augmenting step, we have the final_prompt = $P + KG_{\text{sub}}$.

In summary, FSL, RAG, and GraphRAG are all techniques that aim to enhance the context and increase accuracy for LLM. However, each method has its advantages and disadvantages (cf. Table 2.6), so depending on the specific task, we will choose the appropriate approach.

Chapter 3

Related Work

In this chapter, we first introduce popular tools and frameworks commonly used in traditional analysis. Specifically, we discuss the advantages and disadvantages of existing tools/frameworks. This is crucial because in the *first direction* of this thesis's contribution, we apply traditional analysis. In addition, most state-of-the-art proposals that adhere to traditional analysis-based approaches are developed on top of these tools and frameworks, even though their threat models may vary. For instance, dynamic analysis requires interaction with the application's UI; therefore, automated interaction tools become indispensable. Indeed, these tools/frameworks are often applied in the state-of-the-art presented in Section 3.3 and 3.4, as well as used in our contributions (e.g., MetaLeak, cf. Chapter 5). Therefore, at the beginning of this chapter, we focus solely on the general workflow of the tools and frameworks, rather than a specific threat model (cf. Section 3.1).

As presented in Chapter 1, two of the four main contributions of this thesis, MetaLeak (cf. Chapter 5) and ALIBIS (cf. Chapter 6), revolve around a novel attack vector that poses a risk of leaking sensitive data through EXIF metadata. Therefore, in the following section of this chapter, we first present studies that demonstrate how attackers can exploit EXIF metadata to gather users' personal information. From there, we identify five types of sensitive data contained in EXIF metadata that we will focus on in this thesis. Then, we discuss current solutions that mitigate EXIF-related security risks (cf. Section 3.2). Next, we review previous works that apply traditional analysis methods, namely, static, dynamic, and hybrid analysis, to assess security and privacy violations in both mobile and wearable ecosystems. Specifically, regarding the mobile ecosystem, we focus on studies related to side-channel attacks (SCA) that lead to the leakage of sensitive data because, in the new threat model we discovered, public storage is utilized as a side channel (cf. Section 3.3). On the other hand, regarding the wearable ecosystem, we concentrate on research related to evaluating privacy compliance with respect to sharing all sensitive categories and data types collected (see Table 2.1) and the destinations to which such sensitive data is sent (cf. Section 3.4).

3.1 Tools & Frameworks for Traditional Analysis

In this section, we categorize the tools and frameworks according to the type of traditional analysis they employ, namely static analysis and dynamic analysis.

3.1.1 Static analysis

Flowdroid [9] is a static analysis framework based on taint analysis for detecting sensitive data leaks in Android apps. In general, Flowdroid analyzes the data flow from **sources** (i.e., where sensitive data originates, such as IMEI, GPS, or contacts) to **sinks** (i.e., locations where the data may be leaked, e.g., SMS or HTTP client). Specifically, unlike Java code, which has a *main()* function as the starting point of the app (i.e., the starting point of the data), Android has multiple entry points, such as *onCreate()*, *onStart()*, and *onResume()*. Therefore, first, Flowdroid creates a dummy main method that simulates the order of callbacks in the app's lifecycle to properly understand how the app works. Following, Flowdroid performs source and sink labeling. This labeling process is done manually by configuring the source and sink list in a file called *SourcesAndSinks.txt*. For example, if the sensitive data is GPS, the source might be the *getLocation()*, whereas the sink could be *HttpURLConnection.send()*. If a link is found between the source and sink of sensitive data, the app is considered to be leaking sensitive data. Although Flowdroid achieves 93% recall and 86% precision when benchmarked with Droidbench, it still has some weaknesses as follows. First, Flowdroid has a slow analysis speed and requires a lot of RAM, especially for large size and complex apps. Second, manually labeling sources and sinks leads to inversely proportional scalability and accuracy of Flowdroid. Particularly, the number of apps is very large and highly diverse in function, making it impossible to manually label each one. On the other hand, using the default *SourcesAndSinks.txt* file provided by the developers may lead to missing sources and sinks, which increases the risk of false positives.

3.1.2 Dynamic analysis

Taintdroid [17] is a dynamic analysis system that tracks sensitive data using taint analysis. Essentially, Taintdroid is not an independent tool or framework, but instead, it is an extension of the Android OS that is deeply integrated into the firmware. The Taintdroid system tracks data at multiple levels, including variable-level, method-level, inter-process communication level (IPC-level), and file-level in the Davik Virtual Machine (DVM)[1]. First, the system assigns *"taint tags"* to sensitive data (i.e., sources) such as locations, contacts, and tracks their propagation according to the data flow logic when the data is manipulated or transmitted (i.e., the app executes at run-time). Unlike Flowdroid, sources and sinks are identified automatically based on the native APIs that are predefined in the Android OS kernel; thus, Taintdroid does not require a manually

[1]DVM is a virtual machine designed for Android that runs apps using DEX (Dalvik Executable) format, which is optimized from Java bytecode for better performance on resource-constrained mobile devices.

declared list of sources and sinks in a *SourcesAndSinks.txt* file When tracking sensitive data at the variable-level and method-level, TaintDroid continuously updates the taint tag status after each change in the variables when methods are executed. Additionally, when apps send and receive data from other apps (i.e., IPC-level tracking), the taint tag is also attached to the message. Regarding file-level tracking, if sensitive data is written to the file system, the corresponding file is also tagged with the taint information. All of these measures ensure that sensitive data can be traced and accounted for. Finally, when sensitive data is sent to a sink (e.g., an HTTP client), Taintdroid considers the data to be leaked. The advantage of Taintdoird is that it allows real-time tracking of sensitive data at multiple levels, thus reducing false positives. The system also does not rely on a predefined list of sources and sinks. However, Taintdroid is extremely complex, not easy to install and use because it requires later intervention in the Android OS kernel. In addition, Taintdroid is not compatible with new Android OS versions from version 5.0 because these versions use Android Runtime (ART) as an alternative to DVM.

Besides, many tools that support automatic interaction with the app's GUI are also frequently used in dynamic analysis. Monkey[2] is a command-line tool for generating pseudo-random user event streams, such as clicks, touches, or gestures. While widely used in research (e.g., [33, 62]), Monkey cannot program test scenarios and provides low accuracy if apps require registration or login for deeper functionality. This results in Monkey's interaction coverage being unreliable, and this tool usually gets stuck when navigating complex app UIs. In contrast, Appium[3] supplies a programmable ability for test scenarios. However, its reliance on predefined XML element IDs or names for user interface (UI) interaction limits its scalability [47]. Finally, DroidBot [40] is a UI-guided test input generator employing a dynamically generated state transition model to create inputs based on the UI. However, it has issues with login screens and pop-up windows, which limit its success rate (only 40% in Zhe Liu's research [46]).

3.2 EXIF Metadata Risk & Existing Mitigation Solution

EXIF (Exchangeable Image File Format)[4] was first introduced in October 1995 and has since become a common standard for digital cameras, including smartphones. When users take photos, EXIF metadata is automatically generated and embedded in the image files. EXIF contains a wealth of information about an image, including camera information (brand, model, kernel, or software), image capture settings (ISO, flash, focus), capture date and time, geographical location, and other parameters, as illustrated in Figure 3.1. Thus, EXIF metadata is useful for management, organization, and retrieving images [16]. Thanks to EXIF metadata, two visually identical images can be easily distinguished using hash functions, thus aiding in verifying the image's origin [26]. Therefore, EXIF metadata is still maintained and used despite the numerous security risks associated with it.

[2] https://developer.android.com/studio/test/other-testing-tools/monkey?hl=en
[3] http://appium.io/docs/en/2.19/
[4] https://www.media.mit.edu/pia/Research/deepview/exif.html

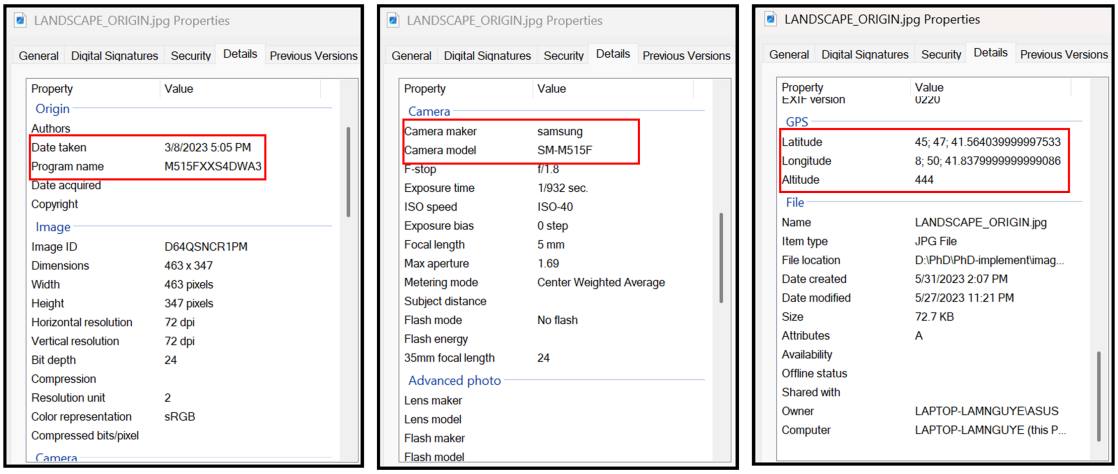

Figure 3.1: EXIF metadata

Indeed, most users are unaware that EXIF metadata contains a significant amount of sensitive data, which can betray users if collected by unauthorized parties, as this information is entirely invisible (cf. Appendix A.2). Many studies have shown that EXIF metadata contains five types of sensitive information that can be exploited by hackers to carry out other attacks (as summarized in Table 3.1), including, datetime, smartphone brand, smartphone model, serial number, and GPS. For brevity, throughout this thesis, we refer to the five types of sensitive information as **sensitive metadata**.

Tayeb et al. [77] indicated the potential for users to be targeted by social engineering attacks by exploiting datetime taken and GPS information when sharing images between social media users.

Charles Gouert et al. [26] highlighted security risks when users share photos containing EXIF metadata, including GPS and serial number. For instance, regarding GPS, users can find themselves in dangerous situations by social engineering attacks if they share a photo of a valuable item they are selling, as potential thieves could extract the image's metadata and determine the location where it is stored. In terms of serial number, users can face spoofing attacks. For example, in 2019, MITRE released a high-severity Common Vulnerabilities and Exposures (CVE) report highlighting a security hole in Amcrest cloud services. The vulnerability is related to the verification mechanism during the association of a new camera with a user's account. Amcrest's authentication process only relies on the camera's serial number without implementing additional security checks to verify ownership. This vulnerability allows hackers to control the camera, for example, turning it on/off or accessing its memory.

Jeremy Faircloth [18] demonstrated that leaking smartphone brands and models through image metadata allows hackers to perform social engineering attacks. Specifically, when hackers know which phone took the picture and its model, they can focus on existing security vulnerabilities on that device. Furthermore, in their research, Rishank Pratik et al. [60] have demonstrated that smartphone brand and model metadata serve

as a means for re-identification attacks.

Table 3.1: Types of attacks associated with sensitive metadata

Sensitive metadata	Attack types
Datetime	Social engineering attacks [77]
Smartphone model	Social engineering attacks & re-identification attacks [18, 60]
Smartphone brand	Social engineering attacks & re-identification attacks [18, 60]
Serial number	Spoofing attacks [26]
GPS	Social engineering attacks [26, 77]

Currently, solutions to mitigate the risk of sensitive metadata leakage can be divided into three groups, as summarized in Table 3.2.

Group 1 proposes using an external tool-based [26] solution (for example, ExifTool[5]) to remove or modify EXIF metadata before sharing images online. Specifically, users can modify the latitude and longitude to provide an approximate location within a specified range (e.g., 10 miles) instead of using the original GPS coordinates. However, tools like ExifTool have two main weaknesses such as (1) ExifTool is executed via a Python script, which is not suitable for most regular users, and (2) ExifTool is designed to be compatible with operating systems such as Windows, macOS, and Linux, and cannot be integrated with mobile apps.

Group 2 proposes using a proxy-based solution [77, 87] to remove EXIF metadata. Specifically, a proxy server will stand between the app and the image recipient (e.g., a social network) to check and clean sensitive data in the image. However, this solution has limitations including (1) implementing a proxy-based solution in Android OS is extremely complicated because Google is concerned that the easy use of proxies can lead to man-in-the-middle (MITM) attacks, so new versions of Android are now equipped with SSL pinning[6] mechanisms to prevent the use of proxies, (2) requiring users to use proxies can lead to suspicion and non-cooperation, and (3) increased costs because servers are needed to deploy proxies.

Group 3 proposed an encryption-based solution [11] to mitigate the risk of sensitive metadata leakage during transmission. However, the solution is primarily dedicated to sharing digital images in the medical field because implementing symmetric and asymmetric encryption infrastructure is expensive, making it less popular for common apps.

[5] https://exiftool.org/
[6] https://developer.android.com/privacy-and-security/security-ssl

Table 3.2: Drawbacks of Existing Solutions for Mitigating EXIF Metadata Leakage

Existing Solutions	Drawbacks
External tool-based [26]	Not supported on Android OS
Proxy-based [77, 87]	• Difficult to implement due to Android's SSL pinning • Users may distrust or refuse to use proxies • High deployment cost due to required servers
Encryption-based [11]	• High cost due to encryption infrastructure • Not widely adopted in regular apps

3.3 Mobile app security and privacy

In this section, we review the existing work related to side-channel attack (SCA) because in our new attack vector, the public storage is exploited as a side-channel due to the security vulnerabilities of the Android OS itself related to the permission model and sandbox mechanism (cf. Section 2.1). Then, we compare these studies in terms of threat model, analysis method, and limitations (cf. Table 3.3).

Side-channel attack (SCA) is a type of security attack that does not directly exploit errors in algorithms or software, but takes advantage of information leaks from the system's execution process. According to the research of Raphael Spreitzer et al. [73], SCA can be classified into two main categories including (1) Hardware-based SCA that mainly exploits physical leaks such as energy consumption, electromagnetic radiation, or hardware behavior (e.g., power analysis attacks or electromagnetic attacks) and (2) Software-based SCA leverages logical weaknesses (e.g., cache attacks, timing attacks), vulnerabilities in the system and protection mechanisms (e.g., permission model and sandbox mechanism), or runtime behavior (e.g., system call/API call/SDK integrated and write/read file) to collect sensitive information (i.e., this type of SCA does not necessarily require direct physical hardware access). In this thesis, we mainly focus on Software-based SCA.

Raphael Spreitzer et al. [74] presented ScanDroid, a hybrid analysis framework for automatically analyzing side-channels present in Java-based Android APIs. ScanDroid aims to detect Android APIs that can be exploited to leak personal user information even without explicit permission. ScanDroid consists of three main modules, namely, Parser, Controller, and Analyzer. Firstly, the Parser module collects a list of Java methods from the official Android Developers website. Then, based on the collected list, the Parser will screen and retain only the methods used to access data (aka accessor methods) based on important prefixes, including *"get"*, *"has"*, *"is"*, and *"query"*. In addition, the Parser also collects all APIs with normal-level permissions, as these APIs do not require explicit user consent. Secondly, the Controller is responsible for automatically triggering events of interest and recording the system's response. Specifically,

ScanDroid focuses on three events that pose a risk of leaking sensitive data through side-channels, including (1) Website Fingerprinting (i.e., leaking the website URL that the user wants to access); (2) Google Maps Search Inference (i.e., leaking the location that the user wants to visit); and (3) Application Start Inference (i.e., collecting a list of currently running apps on a mobile device). ScanDroid uses ADB[7] to interact with the Android device for browsing websites, performing location searches on Google Maps, and launching apps. Then, ScanDroid uses Java Reflections[8] to call methods, constructors, and APIs from the list collected by the Parser and records the execution time series (aka trace) for each action. For example, when a user access two websites, *google.com* and *youtube.com*, and uses an API that can be exploited by a side-channel, *TrafficStats.getTotalRxBytes()*, the Controller will record two traces $T_{\text{train}_1} = \{\text{google.com}, [500, 510, 515]\}$ and $T_{\text{train}_2} = \{\text{youtube.com}, [500, 1200, 1500]\}$. T_{train_1} and T_{train_2} are considered the training set. Thirdly, the Analyzer will analyze the traces to determine the risk of data leakage through side-channels. Specifically, ScanDroid utilizes Dynamic Time Warping (DWT)[9] and the Python framework scikit-learn to determine the shortest Euclidean distance between the test set traces (to be labeled) and the training set traces. For example, suppose a user continues to visit the website google.com multiple times and records T_{test_a}, T_{test_b}, and T_{test_c}, which have a high similarity with T_{train_1} (i.e., a low Euclidean distance). In that case, it can be inferred that the user is visiting the website *google.com* and the API *TrafficStats.getTotalRxBytes()* can be exploited as a side-channel. The ScanDroid is an effective framework for evaluating the leakage of sensitive data through side-channels. However, it has some limitations that reduce the application of this framework at scale. Firstly, ScanDroid primarily focuses on Java-based APIs, while there are many programming languages, such as Kotlin, Flutter, and Native [61], which are used for Android app development. Secondly, ScanDroid apps require direct installation on smartphones and consume a tremendous amount of RAM and time (up to 7-8 hours) when examining the entire API. Lastly, the automated interaction process with the device through the ADB command is only suitable for simple contexts, such as website queries, Google Maps searches, or app launches.

Sajjad Pourali et al. [59] developed ThirdEye (dynamic analysis) to assess the risk of sensitive data leakage in Android apps through side-channels and covert channels. Specifically, the authors focus on three threat models including (1) On-path network attacker: an attacker can intercept and decrypt network traffic between an app and a server if weak keys/encryption (e.g. DES, fixed keys) are used during communication and then obtain authentication tokens to hijack sessions or accounts; (2) Co-located app attacker: an attacker can trick a user into installing a malicious app, then access and decrypt important files of other apps if these apps use encryption algorithms, thereby collecting personal data; and (3) device-owner attacker: where the attacker is the device owner, who wants to access protected content of a service provider, for example, free ac-

[7]Android Debug Bridge (ADB) is a command-line tool that supports communication between a computer and an Android device https://developer.android.com/tools/adb.

[8]Java Reflection is a technique that allows access to classes, methods, properties at run time.

[9]DTW is an algorithm used to compare two time-based datasets to find similarities https://github.com/halachkin/cdtw.

cess to premium/paid services from the app provider. ThirdEye has four main modules, including, Device Manager, UI Interactor, Operations Logger, and Data Flow Inspector. Firstly, the Device Manager sets up the environment for testing. This module utilizes the ADB command to uninstall the app from the last test. After that, it installs the new app and grants all the required runtime permissions. Secondly, UI Interactor performs interactions with the app's UI. Specifically, this module searches for UI components such as buttons and input fields based on a predefined list of English keywords. In addition, this module is equipped with the Google Translate API to translate the app's UI into English in case the app does not support this language. Next, UI Interactor enters input fields using a predefined list of input values, while also triggering buttons based on priority (for example, the *"click"* button has higher priority than the *"not now"* button). In addition, for login/signup forms, UI Interactor prioritizes login via Google account. Thirdly, Operations Logger captures all network traffic (e.g., HTTP and HTTPs protocols), records file access activities (e.g., open and move file APIs), and logs encryption operations (e.g., APIs belonging to the javax.crypto.Cipher class). Finally, Data Flow Inspector analyzes the data collected by Operations Logger to detect security and privacy risks. Regarding privacy, Data Flow Inspector will extract sensitive data (e.g., contacts, messages, images, audio, and videos) stored on the device, then create copies of this information in different encoding formats (e.g., Base64, hex) and compare them with the collected network traffic. Regarding the security issue, Data Flow Inspector verifies whether data is exchanged using unencrypted protocols (e.g., HTTP) or employing weak encryption algorithms (e.g., RC4) or hard-coded keys in the source code. Regarding the detection of side-channels and covert channels, the module checks files in shared storage that are accessed by multiple applications. If apps access the same file path, the module flags potential side-channels and covert channels. ThirdEye was applied to 12,598 apps and detected a number of security and privacy risks that violated the threat model that the author aimed at, such as (1) 2,383/12,598 apps leaked information used to track users, such as (Advertising ID, email,) and (2) 299 apps exchanged using unencrypted protocols (HTTP). In addition, related to side-channels and covert channels, ThirdEye detected (1) 44 apps saving device information to public storage, then 104 other apps checking for the existence of these paths; (2) 4 apps writing the WiFi MAC address to .cc/.adfwe.dat and 8 other apps checking for the existence of this file and (3) 20 apps saving the MD5 hash of the WiFi MAC address and 67 other apps checking for the existence of the path. However, ThirdEye has some disadvantages including (1) it cannot handle apps that do not support registration/login via Google account and require email/SMS authentication, (2) it cannot handle code obfuscation cases because ThirdEye tracks API based on method names; (3) it only supports Java-based programming apps; and (4) it cannot handle apps that use UI animations and images.

Joel Reardon et al. [62] combined static and dynamic analysis to search for security vulnerabilities through side-channel and covert-channel in 88,113 Android apps. Specifically, the threat model focuses on apps and third-party libraries that exploit side-channels and covert channels to collect location, personally identifiable information (PII) such as email, phone number, user ID, and device identifiers (e.g., MAC address, IMEI).

To achieve this, the authors developed an analysis pipeline comprising three stages. In the first stage, the authors built a script to automatically download APK files from the Google Play Store. In the second stage, the authors prepared an environment for dynamic analysis to monitor the app's behaviour at three levels, including (1) platform, (2) kernel, and (3) network. At the platform level, the analysis pipeline integrates monitoring tools into Android 6.0.1 (Marshmallow) to log all resource access activities when the app is installed and running. Particularly, at the kernel level, the authors modified the Linux kernel to record access and share file events. This method enables the pipeline to capture activity logs when the app reads, writes, or shares files. At the network level, the pipeline uses a network monitoring tool based on the Android VPN API to intercept and analyze all sent-out traffic. Additionally, the dynamic analysis environment employs Monkey to interact with the app's UI automatically. In the final stage, the authors examined the logs and network traffic gathered in Stage 2 to identify evidence of side-channels and covert channels. Specifically, after detecting suspicious apps through dynamic analysis, the pipeline performed static analysis to determine how they break the permission mechanism. Rather than analyzing each app individually, the pipeline focused on circumvention techniques originating from specific SDKs, which were identified by the destination endpoints of the network traffic. Specifically, the authors used APKTool to decompile APK files and extract Smali bytecode from suspicious apps, looking for strings containing PII and data sources. Then, they searched for strings related to network destinations by examining logs recorded by platform-level and kernel-level tools. Finally, the authors created a unique fingerprint to identify the presence of vulnerabilities in each SDK, then scanned the entire set of investigated apps (88,113 apps) for this characteristic string to identify other apps that are capable of exploiting the same side-channel and covert-channel, even if they were not initially detected by dynamic analysis. The research identified several security risks, including (1) apps collecting addresses of connected WiFi base stations from the ARP cache (which can be used as location information), (2) apps collecting the MAC addresses of devices using the *ioctl* system call, (3) third-party libraries provided by two Chinese companies, Baidu and Salmonads, using the SD card as a covert-channel to read the phone's IMEI, (4) one app using EXIF metadata to leak location (which is the focus of our contribution, Metaleak, cf. Chapter 5). However, the proposed system still has some weaknesses. Specifically, the analysis pipeline utilizes Monkey to simulate automated user interactions with the apps; thus, it encounters the limitations we outlined in Section 3.1. Furthermore, to assess apps leaking location information through image metadata, the authors examined apps sending GPS information to untrustworthy third-party domains for advertising purposes, rather than investigating if the send-out traffic contains metadata. This examination method might miss many apps and yield false negative results because (1) apps could transmit sensitive metadata within traffic to legitimate domains and pass through the inspection mechanism of the study (as authors only observe untrusted domains) or (2) the image recipients might not be domains but valid alternative endpoints like cloud storage or social networks which used by users. However, these image receivers could retrieve location information from the image and exploit this sensitive data as a side-channel. We

suppose this is why the authors identified only one app leaking location information through EXIF metadata. In contrast, we demonstrated that leakage of EXIF metadata is associated with numerous popular apps (cf. Chapter 5).

Zikan Dong et al. [15] identified security and privacy risks when third-party SDKs use public storage as a side-channel to monitor users' behavior. Specifically, in the past, third-parties often used identifiers such as GAID (Google Advertising ID) or IMEI (International Mobile Equipment Identity) to track and collect user behavioral data. However, after Google implemented several measures to restrict access to hardware identifiers in order to protect user privacy, third-party SDKs employed alternative methods to continue tracking users. In particular, a common technique is that third-party SDKs will automatically generate and store custom identifiers on public storage, allowing multiple apps that use the same SDK to access and identify users. Because public storage in Android is poorly managed (cf. Section 2.1), these identifiers can persist for a long time, posing a risk of policy violations and privacy threats. To investigate the above threat model, the authors developed a dynamic analysis framework with four modules: (1) dynamic analysis, (2) candidate finding, (3) candidate attributing, and (4) static analysis. Specifically, the dynamic analysis module uses Fastbot[10], a Monkey-based tool, to automatically interact with the app's UI. Then, the authors customized the Android kernel to integrate kernel-level instrumentation to record all system calls related to file operations (e.g., read, write) performed by the app. Next, the candidate finding module is responsible for analyzing the logs collected by the dynamic analysis module. Specifically, the authors built a set of rules to automatically detect files potentially containing user identifiers (aka suspicious files). First, in terms of storage location, suspicious files are usually stored in public storage rather than in app-private directories, which are access-restricted, allowing them to be accessed by many other apps. Second, in terms of persistence, suspicious files usually persist across app uninstallations and are rarely deleted, similar to hardware identifiers (e.g., IMEI, serial number, GAID). Third, in terms of cross-application access, suspicious files are often accessed by multiple apps using the same third-party SDK. Next, after identifying suspicious files, the candidate attributing module uses Frida[11] to find the APIs that interact with these files, thereby determining the origin of the API (i.e., which SDK invoked them and the corresponding third-party vendor). Finally, the static analysis module decompiles the app's APK to verify whether the app indeed integrated the third-party SDK identified by the candidate attributing module and analyzes in depth how the identifiers are generated and stored. The authors applied the framework to 8,000 Android apps and identified 17 third-party SDKs that stored identifiers in public storage to track users. Regarding limitations, the research was affected by Monkey's limitations when interacting with the app's UI.

Table 3.3 summarizes the state-of-the-art (SOTA) related to sensitive data leakage caused by software-based SCA that we have reviewed so far.

[10] https://github.com/bytedance/Fastbot_Android

[11] Frida is an open-source toolkit that allows embedding JavaScript into running processes for real-time API hooking, debugging, and inspection.https://frida.re/

Table 3.3: SOTA proposals related to sensitive data leakage caused by software-based SCA

SOTA	Threat model	Analysis method	Limitations
ScanDroid [74]	Android APIs are exploitable for personal data leakage without permission	Hybrid analysis	• Focuses only on Java APIs • High RAM/time usage (7–8h) for full scan • Limited to simple test scenarios using ADB command
ThirdEye [59]	• On-path attacker intercepts weakly-encrypted traffic to steal tokens • Co-located app attacker decrypts other apps' files to steal personal data • Device-owner attacker bypasses protections to access premium content	Dynamic analysis	• No support for apps requiring email/SMS authentication • Fails with obfuscated code (method-name tracking) • Java-only app support • Cannot handle UI animations and images
Joel Reardon's framework [62]	Apps or third-party libraries exploiting side/covert channels to collect location, PII, and device identifiers (e.g., MAC, IMEI)	Hybrid analysis	• Relies on Monkey for app interaction, inheriting its limitations • Only inspects GPS leaks to untrusted domains, leading to false negatives for legitimate domains or other endpoints • Non-comprehensive assessment of EXIF metadata leakage
Zikan Dong's framework [15]	Exploiting public storage to generate identifiers to track user behavior	Hybrid analysis	• Relies on Monkey-based tool for app interaction, inheriting Monkey's limitations

3.4 Wearable app security and privacy

In this section, we review studies that assess privacy violations associated with sharing sensitive data categories and their corresponding destinations, as this is the primary focus of the fourth contribution of the thesis. We then compare the studies in aspects including threat model, analysis method, sensitive data categories, and limitations (cf. Table 3.4).

Doguhan Yeke et al. [90] developed FlowFinder, a static analysis-based tool, to analyze permission model inconsistencies between Wear OS and Android OS. Specifically, the authors found that runtime permissions operate independently on wearable devices and smartphones, but sensitive data flows can be shared simultaneously across both devices. This leads to sensitive data being collected without explicit user consent, especially when users are victims of cross-device permission phishing. For example, users only grant location access to Android apps and do not provide this permission to Wearable apps, but GPS data can still be transferred from a smartphone to a smartwatch via Bluetooth. To identify the aforementioned threat model, FlowFinder performs taint analysis based on Flowdroid. Specifically, since Flowdroid requires a list of sources and sinks as input to analyze an APK, the authors must first manually identify these elements. To identify the sources, the authors use the official Android documentation to extract all APIs protected by dangerous-level permissions. To identify the sink, the authors analyze the Wear OS Data Layer to determine all APIs that could be used for inter-device data transfer. Next, Flowdroid checks for the existence of a data flow from the source to the sink. If a link exists between the source and the sink, the app potentially leaks sensitive data. However, to mitigate False Positives (i.e., Flowdroid identifies the app as leaking sensitive data when it actually does not), FlowFinder extracts the permissions requested by the app from its manifest file. Then, FlowFinder checks whether an app has actually requested the necessary permissions to access sensitive data in the detected flows. If the app does not request the required permissions, the data flow is considered impossible and will be discarded. For example, the *getLatitude* method requires the location permission. If FlowFinder detects a data flow originating from *getLatitude* but the app hasn't requested location permission, that flow is ignored. The authors tested FlowFinder on 150 apps and discovered that 28 of them contained cross-device sensitive data flows between smartphones and smartwatches. In addition, because FlowFinder is built on top of FlowDroid, it inherits the tool's shortcomings, most notably high memory usage and limited scalability.

Babatunde Olabenjo et al. [57] applied hybrid analysis to analyze the risk of sensitive data leakage in wearable apps. First, the authors conducted static analysis using the AndroWarn tool to detect risky behaviors and potential information leakage inside the app. AndroWarn is based on a taint analysis similar to Flowdroid. Given an input APK file, AndroWarn parses the Manifest file from an APK file and collects all dangerous-level permissions. Next, it decompiles the APK file to obtain all classes.dex files, which contain Dalvik bytecode. From there, AndroWarn looks for sources (points where sensitive data originates, such as GPS, contacts, or serial numbers) and sinks

(points where this data could potentially leak, such as HTTP clients or SMS). It analyzes the dex files to detect possible links between sources and sinks. For instance, a connection between the API *LocationManager.getLastKnownLocation()* (source) and *HttpClient.execute()* (sink) would indicate that location data might be sent out over the internet. Finally, AndroWarn maps the source and sink with dangerous-level permissions. For example, the permission *ACCESS_FINE_LOCATION* corresponds to the source *LocationManager.getLastKnownLocation()*, while *INTERNET* permission corresponds to the sink *HttpClient.execute()*. If both the source–sink link and the associated permissions are present, the app is labeled as posing a risk of sensitive data leakage. Specifically, in this example, the app is considered to be leaking location data via an internet connection. Second, the author performs dynamic analysis to verify the results of static analysis. Specifically, each app will be executed for 15s and use Monkey to automatically interact with the app to simulate realistic runtime behavior. At the same time, the authors use MITM proxy to intercept and extract network traffic. The results of applying the above hybrid analysis process to 4,017 apps (1,894 companions, 1,894 embedded, 229 standalone) show that companion apps leak the most sensitive data (909/1,894), while embedded apps (276/1,894) and standalone apps (21/229) leak the least. Regarding limitations, because the authors rely on AndroWarn (similar to Flowdroid) for static analysis and Monkey for dynamic analysis, the proposed solution also inherits the weaknesses of the two tools mentioned above (cf. Section 3.1).

Zeya Tan et al. [76] proposed PTPDroid, a static analysis-based framework that checks the consistency between privacy policies and data flows to detect violations of disclosing user data to third parties as outlined in the privacy policy. Specifically, PTPDroid receives the APK file and the corresponding privacy policy of the app as input and then determines the level of violation including (1) vague disclosures (i.e., only providing generic descriptions of the types of data shared and the third parties receiving it), (2) omitted disclosures (i.e., not declaring the behavior of sharing user data to third parties), and (3) incorrect disclosures (i.e., the privacy policy denies sharing data with third parties but this behavior actually occurs in the app). PTPDroid consists of three main stages: (1) ontology and mapping establishment, (2) privacy policy analysis, and (3) static analysis. Specifically, in the ontology and mapping establishment stage, PTPDroid standardizes and maps strings (e.g., API, URI, domain) in the app source code to sensitive data or third-party entities. For example, the API *getLastKnownLocation()* is mapped to sensitive data (i.e., location), the URI *content://com.android.calendar* is mapped to sensitive data (i.e., calendar), and the domain *http://app.adjust.com* is mapped to the third party (i.e., Adjust). This mechanism facilitates identifying the types of data that are collected by the app and which third parties receive them. Second, in the privacy policy analysis stage, PTPDroid converts the privacy policy into a tuple structure (`<actor, action, data type, entity>`) for easy comparison with actual behavior in the app code. For example, the sentence *"We may share your location with advertisers"* will be converted to `<we, share, location, advertiser>`. Finally, in the static analysis stage, PTPDroid identifies sensitive data flows in the app using Flowdroid (i.e., from the API that receives user data to the API that sends data

out) and normalizes them into a tuple structure (`<entity, action, data type>`) for comparison with the privacy policy. For example, static analysis identifies the existence of a link between the source API, *getDeviceId()*, used to retrieve the device ID, and the sink API that sends data to Facebook, *sendDataByPost(https://www.facebook.com)*. The result after normalization is a triple `<Facebook, collects, device ID>`. When applying PTPDroid to 1000 apps, the authors obtained 3,351 sensitive data flows from 947 apps to 88 third parties. Of which, device ID is the most leaked data type, accounting for 87.5% of sensitive data flows, while location is the second most leaked data type, accounting for 9.2% of data flows. Additionally, 71.9% of apps were categorized as vague disclosures, 17.8% as omitted disclosures, and 1.2% as incorrect disclosures. Finally, since PTPDroid is built on top of Flowdroid, it also inherits the weaknesses of this tool.

Huajun Cui et al. [12] developed TraceDroid, a new framework for analyzing network traffic to assess the risk of sensitive data leakage to third-party services (TPS). The authors found that network traffic analysis is a complex problem due to the limitations of existing tools. Specifically, TCPDUMP cannot handle encrypted traffic (e.g., HTTPS), while the MITM proxy has difficulty if the traffic is mixed (e.g., operating system traffic, app backend traffic, and TPS traffic). In addition, existing methods mainly rely on predefined TPS lists (i.e., whitelists) to identify TPS. Therefore, when many new TPS are introduced, this approach suffers from low accuracy because the whitelists are often outdated. TraceDroid consists of three phases, namely (1) network hooking, (2) traffic triggering, and (3) TPS identification. In the network hooking stage, TraceDroid doesn't try to hook all networking libraries separately. Instead, it focuses on the OpenSSL library and targets just two key APIs: *SSL_Read* (used to read raw data after receiving and decrypting HTTPS traffic) and *SSL_Write* (used to write raw data before encryption and transmission) By this approach, TraceDroid can access traffic in plaintext form. Additionally, TraceDroid hooks the default Android APIs for socket connections: *socketWrite0* and *socketRead0*. Next, in the traffic triggering stage, the author utilizes a self-developed tool called obsCleaner to remove interfaces such as user agreements, privacy policies, and splash screens in the app to access deeper features. After that, they continue to use AutoClick (a tool similar to Monkey) to automatically interact with the app's UI and generate sent-out traffic. In the TPS identification stage, the authors build an unsupervised method to identify TPS without a whitelist. Specifically, based on the network request log, TraceDroid only retains domains accessed by at least 10 apps. For example, the domain *graph.facebook.com* was accessed by 45 apps, while *api.myweather.com* was accessed by only 1 app, indicating that the former belongs to a TPS, whereas the latter belongs to an app's backend. Next, TraceDroid groups similar network requests into the tuple `<protocol, method, host, path_pattern, key_list>`. For example, the network log has two requests: (1) `GET https://graph.facebook.com/send?advertiser_id=ABC123` and (2) `GET https://graph.facebook.com/send?event=fb_mobile_purchase`. Then, the tuple is `<HTTPS, GET, graph.facebook.com, /send, {advertiser_id, event}>`. Finally, the tuple is reduced to identify the TPS (in this case, Facebook). The authors

applied TraceDroid to 9,771 apps and identified apps that leaked sensitive data up to 357 TPS. Specifically, 39% leaked device information (e.g., IMEI, Serial Number), 45.8% leaked location information (GPS), and 42.6% leaked network information (e.g., WiFi state, IP address). On the downside, TraceDroid can only investigate network libraries (i.e., OpenSSL) and default Android APIs (i.e., SSL_Read/SSL_Write and socketWrite0/socketRead0). This can easily miss apps that also leak sensitive data but do not use the default libraries and APIs. In addition, identifying TPS based on a threshold of the number of apps accessing a domain (i.e., 10 apps) may also lead to missing some TPS. Furthermore, TraceDroid does not consider the case where an app may send sensitive information to its backend. Finally, TraceDroid focuses on only three data categories: device information, location, and network information, whereas Android encompasses 14 categories of sensitive data (cf. Table 2.1).

App developers often use third-party SDKs to accelerate the app development process, rather than building them from scratch. However, this trend poses a risk of leaking sensitive user data, as developers often do not fully understand the privacy settings of the SDKs. Consequently, they usually keep the default configurations, which are more inclined to collect data than to provide security functions. In their work, D. Rodriguez et al. [63] focus on identifying the aforementioned security and privacy risks associated with Facebook SDKs, specifically the Facebook Android SDK and Facebook Audience Network SDK, due to the widespread use of these SDKs. The authors have built an analysis process consisting of 3 stages: (1) static analysis, (2) dynamic analysis, and (3) compliance Analysis. First, static analysis inspects the app's source code to determine whether it integrates the Facebook SDK, and, if so, verifies whether the privacy settings remain at their default values or have been modified. Specifically, the authors use LibScout to identify third-party SDKs that are included in the app. LibScout remains effective even when the code is obfuscated, as it generates unique SDK *"fingerprints"* from stable features such as API calls, class structures, and method signatures, and then matches these fingerprints against the app's source code Next, the authors use Apktool to collect the app's Manifest file and parse XML tags to detect changes related to the Facebook SDK's privacy settings made by the app developer. Second, the dynamic analysis aims to observe the actual behavior of the app at runtime, including the types of sensitive data transmitted and modifications of privacy settings (i.e., overwriting privacy settings instead of using configurations from the Manifest file). The authors install the app on an Android phone and use Monkey to interact with the app's UI automatically. Next, they use MITM proxy to intercept and analyze network traffic in order to detect sensitive data being transmitted. In addition, Frida is used to inject scripts to retrieve the values of getter and setter functions in the SDK to determine the runtime configuration of privacy settings, for example, the setter function, setAutoLogAppEventsEnabled(true) means allowing the SDK to send data to Facebook. Lastly, compliance analysis compares the results obtained by static and dynamic analysis with the list of sensitive data types that the app will collect and share, as declared by the app developer in the data safety section. The authors applied the above analysis process to 6,000 apps and found that 83.23% of the apps set the default value for privacy settings (i.e., allowing the SDK

to collect data). Additionally, 3,589/6,000 apps share user data through the Facebook SDK, with the device model and AdID (Advertising ID) being the two most commonly shared data types. Regarding privacy compliance, 399 apps collected AdID but did not disclose it in their Data Safety policies. In terms of weaknesses, this work focuses on only one specific case, the Facebook SDK. In addition, LibScout was shown to be less effective at identifying Facebook SDK integrations, especially with newer versions, due to outdated fingerprints and limited database coverage. Finally, the analysis process also inherits the weaknesses of the Monkey tool.

Table 3.4 summarizes the state-of-the-art (SOTA) related to wearable ecosystem privacy non-compliance we have reviewed so far.

Table 3.4: Comparison of SOTA proposals on sensitive data leakage and their limitations.

SOTA	Threat model	Analysis method	Sensitive data categories	Limitations
FlowFinder [90]	Sensitive data leak due to an inconsistent permission model between Wear OS & Android OS.	Static analysis	• Location • Device or other IDs	Inheriting FlowDroid's limitations
B. Olabenjo's framework [57]	Sensitive data leakage in sent-out traffic.	Hybrid analysis	• Location • Device or other IDs • App info & performance • Photos or video • Audio files	Inheriting the limitations of FlowDroid and Monkey
PTPDroid [76]	Privacy violations: discrepancies between actual sensitive data sharing & declared Data Safety.	Static analysis	• Device or other IDs • Personal info • Location • Contacts • App activity	Inheriting FlowDroid's limitations
TraceDroid [12]	Leakage of sensitive data to TPS.	Dynamic analysis	• Location • Device or other IDs	• Missed violator apps due to focus on default libraries/APIs only • Missed TPS due to domain-access threshold • Ignores leaks to app's own backend
D. Rodriguez's framework [63]	Leakage of sensitive data due to Facebook SDK integration.	Hybrid analysis	• Device or other IDs	• Only focuses on Facebook SDK • Inheriting Monkey's limitations

Chapter 4

LLM on support of privacy & security of mobile apps

In this chapter, we present recent studies that apply LLM-based methods to analyze security and privacy risks in mobile ecosystems. Indeed, the positive results achieved by these works demonstrate that LLM has the potential to support and gradually replace the traditional analysis-based approach. This constitutes the *second direction* of contribution in this thesis. Firstly, we outline the top 10 security and privacy issues associated with mobile apps according to the OWASP report. OWASP (Open Web Application Security Project)[1] is an international non-profit organization specializing in web application security. However, in recent years, this organization has expanded its scope to include mobile apps, aiming to raise awareness of security/privacy vulnerabilities. Secondly, we present state-of-the-art research on applying LLM to the security and privacy aspects of mobile apps. Since several of the surveyed research proposals can be applied to analyze one or more OWASP vulnerabilities, we categorize them based on their target applications. More precisely, we classify the analyzed state-of-the-art research proposals into three groups: *vulnerabilities detection (Section 4.2)*, *bug detection and reproduction (Section 4.3)*, and *malware detection (Section 4.4)*. Although the application goals differ, all these research proposals leverage LLM to analyze the app source code. One of their main contributions is how to optimize the input prompt for LLM to target the specific scenario being addressed. Indeed, as discussed in Section 2.3, the input prompt plays a decisive role in the performance of LLM. The main difference between the analyzed research proposals is how the authors collect information to enhance the context for the input prompt. Finally, at the end of this chapter, we provide a table 4.3 summarizing the reviewed studies that use LLM-based approaches to analyze mobile app security/privacy. For each paper, we specify the LLM used, the research objectives, and the corresponding OWASP risks it addresses.

In summary, this chapter contributes two main points to the thesis. First, we provide a comprehensive assessment of the ability of LLM to identify common security vulner-

[1] https://owasp.org/

abilities of the mobile ecosystem. This establishes a key justification for the choice of LLM to gradually replace the traditional methods presented in Chapter 3. Second, and the main focus of this contribution, we conduct a structured state-of-the-art literature review, focusing on the application of LLM and the challenges that existing studies face when using this technology. A key finding synthesized in this chapter is the crucial role of prompt engineering and context enrichment in enhancing the performance of LLMs for specific security scenarios. This synthesis not only highlights the potential of LLM to complement or replace traditional methods, but also helps us position approaches for the specific contributions of ALIBIS (cf. Chapter 6) and WearLeak (cf. Chapter 7).

4.1 OWASP Mobile Top 10

In this section, we introduce the causes of the top 10 OWASP risks and their practical exploitability levels, as reported by OWASP.[2] Then, we summarize the impact of each vulnerability in Table 4.1

Risk 1 (R1): Improper Credential Usage (Exploitability level: Easy):
Some developers have the dangerous habit of hardcoding authentication information (e.g., secret key, password, API key, etc.) into the app's source code. This makes the software development process faster and more convenient for developers instead of implementing dynamic key storage solutions such as OAuth, JWT, vault, etc. However, it causes serious vulnerabilities because hackers can easily decompile APK (Android Application Package) or iPA (iOS App Store Package) to collect credential information and illegally access the private accounts of software developers and users.

R2: Inadequate Supply Chain Security (Exploitability level: Average): This security risk refers to hackers tracking activities and stealing sensitive user data by exploiting security vulnerabilities when the app integrates with third-party libraries, APIs, or SDKs (Software Development Kits). This attack allows hackers to insert malicious code, install spyware, or steal credential information.

R3: Insecure Authentication/Authorization (Exploitability level: Easy): When hackers identify a vulnerability in an authentication or authorization mechanism, they can impersonate a user and bypass the login to access personal data by sending a request directly to the backend server without going through the identity validation step. Additionally, hackers can achieve higher-level permissions than their actual authorized level to gain unauthorized access to sensitive information.

R4: Insufficient Input/Output Validation (Exploitability level: Difficult): Insufficient validation or sanitization of user-provided inputs, such as registration information or uploaded files, can pave the way for attacks, including SQL and command injection, and cross-site scripting (XSS). Hackers can leverage these weaknesses to steal users data, alter the app's behavior, and break the entire mobile platform.

R5: Insecure Communication (Exploitability level: Easy): Most mobile apps need to communicate with a server (service provider) to function correctly. They also

[2] https://owasp.org/www-project-mobile-top-10/

communicate with each other (for example, a wearable app on a smartwatch periodically communicates health data to a companion app on a smartphone). Hackers can exploit this behavior to eavesdrop sensitive information (sniffing attacks) or interfere with and modify packets in the transmission line, leading to incorrect application functionality (Man-in-the-Middle attacks). The most significant factor contributing to this vulnerability is that mobile apps frequently communicate with one another in plain text rather than delivering data using encryption protocols such as SSL/TLS. In addition, wrong implementation of SSL/TLS, such as using self-signed, revoked, or expired certificates, also leads to insecure communication.

R6: Inadequate Privacy Controls (Exploitability level: Average): Android OS applies a permission model to protect user privacy. Run-time permissions[3] enable users to authorize mobile apps to access sensitive information (e.g., emails, credit card details, contacts, health data, GPS) or utilize the hardware of the mobile device (e.g., camera, microphone, health sensors, location sensors, Wi-Fi, Bluetooth). Hackers can exploit the vulnerabilities of the permission model to access personal information or device resources without explicit consent from the user.

R7: Insufficient Binary Protections (Exploitability level: Easy): Binary files (e.g., APK files or iPA) contain a lot of sensitive information, such as credentials information or app logic that can br used to infer business strategies. Binary files are faced with two main attack types, namely reverse engineering and code tampering. In reverse engineering, hackers decompile the bìnary files to inspect the app's source code and then search for valuable information. In addition, by code tampering, hackers can remove license-checking code to use advanced features as a premium user. Therefore, a binary file that lacks proper protection can cause damage to both users and developers.

R8: Security Misconfiguration (Exploitability level: Difficult): Mobile apps provide a set of rules to configure permissions and security settings; for example, Android uses the *AndroidManifest.xml* file, and iOS uses the *Info.plist* file. In addition, Google and Apple also provide a list of permissions necessary for the app's features, and each permission has a specific scope. The developers are responsible for configuring the permissions to suit the application's functionality. However, misconfigurations are possible. In addition, the combination of permissions can lead to unexpected results beyond the developer's control if they do not fully understand the meaning of each permission. Hackers can take advantage of this confusion to attack users.

R9: Insecure Data Storage (Exploitability level: Easy): Stealing users' personal information and sensitive data stored in insecure storage is straightforward. For instance, Android OS provides a sandbox mechanism to segregate data among apps to counter this threat. This mechanism creates a secure and restricted environment for each app, ensuring that they operate independently and do not interfere with others. File isolation is the most important feature of the sandbox mechanism. However, Android OS has three storage classes, namely system class, application-specific class, and public storage class, and not all classes are protected by the sandbox mechanism. Specifically, the system class, where the entire OS is stored, is protected by Linux access control,

[3] https://developer.android.com/training/permissions/requesting

whereas the application-specific class is protected by the sandbox mechanism. In contrast, the public storage class (e.g., SD cards and other logical partitions that are shared among apps) is solely secured by the permission model. Apps use public storage to store media files, including photographs, music, movies, and documents. This means that any app that is granted read/write storage permissions can read and overwrite information in media files created by other apps. This could lead to the destruction of files or the leakage of sensitive information.

R10: Insufficient Cryptography (Exploitability level: Average): If insecure or outdated encryption algorithms such as MD5[4], DES[5], Triple DES[6], or SHA-1[7], are used in mobile apps, they can lead to breaches of confidentiality, integrity, and authentication of sensitive information. Furthermore, while symmetric encryption algorithms like AES are highly reliable, hackers can exploit them if developers implement them with weak secret key management, such as storing the key directly in the source code.

Table 4.1 lists the top 10 OWASP mobile security risks (R1–R10) together with representative examples of their impacts reported in prior studies.

[4] https://www.ietf.org/archive/id/draft-ietf-tls-md5-sha1-deprecate-09.html

[5] https://www.cisa.gov/sites/default/files/2024-05/23_0918_fpic_AES-Transition-WhitePaper_Final_508C_24_0513.pdf

[6] https://www.nist.gov/news-events/news/2023/06/nist-withdraw-special-publication-800-67-revision-2

[7] https://www.nist.gov/news-events/news/2022/12/nist-retires-sha-1-cryptographic-algorithm

Table 4.1: OWASP vulnerabilities and examples of impacts.

Risk	**Examples of impact**
R1	[96] shows that 51.5% (121/237) of the analyzed apps leaked email service keys, and 67.3% (132/196) leaked Amazon AWS API keys because the developer hardcoded authentication information.
R2	[86] shows that 66% of 100 popular iOS apps leaked WeChat SDK credentials and 37% for Weibo SDK (two popular SDKs with over 40 million users).
R3	[83] shows that 86.2% of 4,000 apps in the Chinese market have deviated implementation of OAuth 2.0 from the standard recommended by RFC.
R4	[10] shows that hybrid Android apps (developed using web programming languages but running on Android OS) are vulnerable to XSS attacks.
R5	[23] shows that – 47.8% of URLs in 303 open-source apps and 69.3% in 3,073 closed-source apps use HTTP (unencrypted). – 67.7% of URLs in open-source apps and 88.3% in closed-source apps lack HSTS implementation (forcing redirect from HTTP to HTTPS).
R6	[54] analyzed 239,381 Android apps, among which: – 30,160 apps shared user data without showing consent popups; – 13,082 apps with popups (23% sent data before user agreement, 8.28% forced agreement (no refuse option), and 1% shared data despite user refusal).
R7	[34] found that image classification apps store AI models (i.e., parameters for deep learning models) directly in the app source code, leading to adversarial attack risks. The author successfully attacked 71.7% of AI models in 114 image classification apps.
R8	[39] shows that Android apps can access device location with only *ACCESS_WIFI_STATE* and *INTERNET* permissions. In 2,089,169 analyzed apps, 18.1% relied solely on these permissions without requesting GPS-related ones.
R9	[62] shows that Baidu and Salmonads SDKs exploited the SD card as a covert channel to access the phone's IMEI (International Mobile Equipment Identity).
R10	[92] analyzed over a million apps, identifying 223 apps signed with a weak 512-bit RSA key and 52,866 apps using the insecure MD5 algorithm.

4.2 Vulnerability Detection

Mobile apps are one of the most dynamic ecosystems, making them a double-edged sword: rapid development is always accompanied by constant security attacks. On the one hand, developers strive to safeguard user security and privacy. On the other hand, attackers continuously search for potential vulnerabilities in various areas, such as the operating system, storage, network, and permission models, among others, and then exploit them to launch complex attack scenarios often beyond the reach of traditional analysis methods. LLM, with its excellent code summarization capabilities, can be used to automatically identify some of those security/privacy vulnerabilities.

Indeed, Kouliaridis et al. [37] evaluated the performance of nine LLM (opensource and paid), namely GPT-3.5, GPT-4, GPT-4 Turbo,[8] Llama-2,[9] Zephyr Alpha,[10] Zephyr Beta,[11] Nous Hermes Mixtral,[12] Mistral Orca,[13] and Code Llama,[14] in identifying the OWASP Mobile Top 10 vulnerabilities presented in Section 4.1. In addition, the authors compared the code summarization capabilities of these nine LLM with two commonly used code analysis tools, namely Bearer[15] and MobSF.[16] Specifically, the authors created a dataset, called *Vulcorpus*, containing 100 code snippets representing the 10 OWASP vulnerabilities, where each vulnerability has 10 code snippets. The authors assessed each LLM's performance across two tasks: (1) detecting vulnerabilities (D score) and (2) providing recommendations to address them (I score). The experimental results showed that the GPT-4 has the best performance on both vulnerability detection (average D score of 6.7) and improvement solutions (average I score of 9.2). In contrast, the Code Llama model achieved the highest average D score (8.1), but it cannot effectively provide solutions for fixing vulnerabilities (an average I score of 4.9). In contrast, Bearer and MobSF detected only 29% and 12% of the code segments containing security/privacy risks, respectively, indicating that LLM perform better than the current popular static analysis tools.

Table 4.2 lists the LLM with the best vulnerability detection performance (i.e., those with the highest D scores) for each specific OWASP vulnerability.

Gabriel Morales et al. [49] leveraged ChatGPT model 3.5 turbo to examine the actual behavior of apps against the developers' published privacy policies. Specifically, the authors perform three tasks: Task 1 - extracting app privacy policies, Task 2 - detecting sensitive data leaks from apps, and Task 3 - identifying non-compliance with privacy policies. The authors collected 200 apps with the highest number of downloads on Google Play and then randomly selected 50 apps from this collection to perform the 3

[8] https://platform.openai.com/docs/models/gpt
[9] https://www.llama.com/llama2/
[10] https://huggingface.co/HuggingFaceH4/zephyr-7b-alpha
[11] https://huggingface.co/HuggingFaceH4/zephyr-7b-beta
[12] https://ollama.com/library/nous-hermes2-mixtral
[13] https://ollama.com/library/mistral-openorca
[14] https://ollama.com/library/codellama
[15] https://github.com/Bearer/bearer
[16] https://github.com/MobSF/Mobile-Security-Framework-MobSF

Table 4.2: Performance of LLM models in detecting specific vulnerabilities [37].

OWASP Risk	**Best LLM**
R1	GPT-4
R2	GPT-3.5
R3	Zephyr Beta, Code Llama
R4	Nous Hermes Mixtral
R5	Zephyr Alpha, Zephyr Beta, Llama-2, Code Llama
R6	GPT-4
R7	Code Llama
R8	Nous Hermes Mixtral, Code Llama
R9	Mistral Orca, Zephyr Beta
R10	Zephyr Alpha

tasks above. In task 1, the authors extracted action verbs and corresponding information types from apps' privacy policies, for example, action verbs including *"collect", "share", "use"*, etc., and corresponding information types such as *"location", "IP address", "device ID"*, etc., and then concatenated them to build a privacy policy repository consisting of 50 policies and corresponding 50 APK files. Next, task 2 performed static analysis. They used FlowDroid[17] to extract sensitive sources (e.g., location, IP address) and their connections to network sinks. Then, the app's source, sink, and connection information is sent to ChatGPT for natural language representation (i.e., code summarization) to determine whether the app shares sensitive data. Finally, in task 3 the author evaluated the semantic similarity between task 1's and task 2's output. First, the author created ground truth data by manually analyzing the privacy policies and sensitive data leaks of 50 apps in the dataset. The author employed two methods to assess the similarity: cosine similarity and the usage of ChatGPT. With the cosine similarity approach, the author transformed task 1's and task 2's output into two embedding vectors (i.e., numeric vectors) and then computed the cosine similarity. Regarding the usage of ChatGPT, the authors use the output of tasks 1 and 2 as input prompts for ChatGPT to evaluate how well the app's data flow aligns with its policy. A score of 1 means the app's policy matches the data flow. In contrast, a score of -1 indicates a contradiction between the policy and the data flow. That means the app collects sensitive data, but the policy states that it does not collect this information. A score of 0 indicates that the app collects sensitive data that are not mentioned in the policy. ChatGPT showed better performance than cosine similarity. Specifically, with the same ground truth, ChatGPT achieved 72.41% accuracy and 83.33% precision, while the cosine similarity approach had 46.7% accuracy and 34.35% precision.

Sahrima Jannat Oishwee et al. [56] examined the potential of LLM to help developers address the complexity of Android permissions (cf. R8 in Section 4.1). Specifically, the paper compared ChatGPT (model GPT 3.5) answers with accepted answers on Stack

[17] https://github.com/secure-software-engineering/FlowDroid

Overflow[18] (a community dedicated to helping developers with programming-related issues) regarding Android permissions. The authors proposed a three-step workflow that includes: (1) data extraction, (2) data processing, and (3) data analysis. In the data extraction stage, all posts about Android permissions are extracted from the Stack Exchange (by limiting the analysis to posts from August 2018 to October 2022 because this period coincides with the release of Android OS versions 9/10, 11, 12, and 13). The authors obtained 1008 pairs of questions and accepted answers. Next, the authors collected information about permission names and corresponding restrictions from Google documentation to form a list of 765 unique permissions for Android 9 - 13. In the data processing stage, questions longer than 4097 words were first truncated due to the token limitation of GPT-3.5, and images were removed from questions if present because GPT-3.5 does not support images. Following that, all questions related to Android permissions were sent to ChatGPT to generate three responses for each question. In total, they obtained 3,024 responses corresponding to 1008 questions. The goal of generating three responses for each question is to evaluate whether ChatGPT provides consistent answers to questions related to Android permissions. Finally, in the data analysis stage, the authors conducted a qualitative analysis by using open coding to evaluate the similarity between ChatGPT answers and Stack Overflow accepted answers. An evaluation team analyzed and categorized ChatGPT answers into three groups, namely, matched, partially matched, and unmatched. Specifically, matched answers have the same meaning as the accepted answers from Stack Overflow, although they may have differences in words, phrases, or descriptions. Conversely, unmatched responses are semantically different from the accepted replies on Stack Overflow. Finally, partially matched answers align with the ideas of the accepted answers but do not provide a suitable solution to help developers solve the complex of Android permission. The analysis showed that 30.75% of ChatGPT's answers were classified as matched, whereas 22.51% were labeled as partially matched.

4.3 Bug Detection and Reproduction

Bugs are unwanted but always exist in the software development process. They not only affect the user experience but also allow hackers to threaten user security and privacy. Bug detection and reproduction are extremely complicated because apps have different logics and are programmed with diverse approaches. Typically, dynamic analysis is applied to interact with the app's GUI to find and reproduce bugs. However, scalability is the biggest weakness of dynamic analysis, as discussed in Section 3.1. Therefore, some researchers analyzed whether LLM's natural language and code understanding capabilities can be used to generate testing scenarios and automatic interactions that can be massively applied to numerous apps.

For instance, Zhe Liu et al. [44] designed GPTDroid, a tool based on ChatGPT model GPT-3.5-turbo for GUI automated testing. The primary objective of GPTDroid is to

[18] https://stackoverflow.com/

tackle existing issues in GUI automated testing, such as inadequate test coverage, limited generalization capability, and excessive reliance on training data. GPTDroid is based on simulating the software testing process as a Q&A task by passing GUI information to LLM to generate and execute test scenarios. At the same time, the apps' responses are sent back to LLM to guide the subsequent actions. This process is repeated until the app is completely tested. As a result of applying GPTDroid to 223 apps, the author discovered 135 bugs related to 115 apps, of which 53 bugs belonging to 41 apps were newly discovered bugs. This result demonstrates that LLM can be a valuable alternative to existing tools (such as Monkey) to improve the scalability of dynamic analysis.

Zhe Liu et al. [45] developed InputBlaster, a ChatGPT-based solution that automatically generates text inputs to detect bugs when users accidentally or intentionally enter unwanted characters. Unwanted inputs can lead to sensitive information leaks, data destruction, app crashes, and even affect the entire backend system of the app on the developer side (cf. Section 4.1). InputBlaster consists of two modules. Module 1 (Prompt Generation for Valid Input) is designed to produce valid inputs that will assist Module 2 (Prompt Generation for Test Generator with Mutation Rule) in generating mutant inputs. Module 1 extracts the hierarchical structure of the app's GUI to collect a list of widgets, mainly concentrating on text-related widgets. Furthermore, it extracts the app's name and list of the app's activity to enhance comprehension of the app's context (e.g., the app's functionality). Subsequently, based on the extracted information, module 1 formulates the relevant constraints for the widget (e.g., the widget requires no special characters or only numbers). Based on the above information, the authors leverage ChatGPT to generate valid input for the widget. This valid input is then entered into the app GUI to get feedback for the valid input. A list that includes widget information, widget constraints, valid inputs, and feedback for valid inputs is sent to module 2 to generate mutant inputs, by providing illustrative examples to ChatGPT. Specifically, the authors collected information about invalid inputs for Android apps reported on GitHub. This information is then used to generate a database vector using the RAG process (cf. Section 2.3) to enrich the context for the LLM. InputBlaster was tested with 36 text input widgets with crashes related to 31 popular Android apps, and the results show that it achieves a 78% detection rate, 136% higher than the best baseline method.

4.4 Malware Detection

Malware is malicious software designed to infiltrate, harm, or disrupt the operation of a computer system, network, or device without the user's permission. Mobile apps, with their diverse functions and complex behaviors, are a favorable medium for hiding malware under legitimate features. Taking advantage of LLM's ability in code summarization to identify abnormal behaviors is a new and promising research direction.

In this area, Wenxiang Zhao et al. [95] designed AppPoet, a system based on model GPT-4 to detect Android malware. AppPoet extracted four types of app features: *permissions*, *API*, *URL*, and *usage features*. The author divided permissions into requested permission and used permission. Similarly, APIs are divided into restricted APIs and

suspicious APIs. Static analysis tools are then used to collect used permissions and restricted APIs and then form mapping relationships between them. Next, information, including permissions, APIs, URLs, and uses-features (i.e., app's functionality), are aggregated to create three views, namely permission view, API view, and URL & uses-feature View. Information about views is then entered into the multi-view text generator module to generate natural language descriptions and summaries of behavior for each view by using multi-view prompt engineering (cf. Section 2.3). The detection classifier module then converts the descriptions and summaries of all three views into three machine-processable vectors and merges them into a single vector describing the APK's behavioral semantic information. In addition, the authors train a DNN-based classification model with a dataset of 11,189 legitimate apps and 12,128 malware apps taken from AndroZoo.[19] The APK's semantic behavioral description vector is then fed into the classifier to predict whether the app is malicious or not. The experimental results show that AppPoet has superior malware detection performance compared to traditional methods such as Drebin (using the string-based method), LBDB (using the image-based method), and MaMaDroid and Malscan (both using the graph-based methods). Specifically, AppPoet achieved an accuracy, precision, recall, and F-1 score of 97.15%, 97.03%, 97.39%, and 97.21%, respectively. Finally, LLM can provide detailed reports related to the malware detection process, e.g., which components of the app are infected, instead of just classifying the app as *"malicious"* or *"benign"*. The authors used the diagnostic report generator module along with descriptions and summaries of the three views of the APK to enhance the context and help the LLM generate human-readable reports.

Table 4.3 reports, for each of the papers described above, the LLM used, the main research targets, and the OWASP risks addressed.

The state-of-the-art research presented above is a valuable starting point for the adoption of LLM to evaluate security and privacy risks in mobile apps. The research results are promising, inspiring us to apply the LLM to the contributions described in Chapters 6 and 7.

[19] https://androzoo.uni.lu/

Table 4.3: Surveyed LLM-based approaches and related OWASP risks.

Category	**Paper**	**Used LLM**	**Research target**	**OWASP Risks**
Vulnerabilities detection	[37]	GPT-4 & Code Llama	Identify vulnerabilities in the app & propose remediation solutions	R1 to R10
	[49]	ChatGPT (GPT-3.5)	Examine the actual app's behavior & its privacy policies	R3,R5,R6, R8 & R9
	[56]	ChatGPT (GPT-3.5)	LLM assist developers in managing complex Android permissions	R8
Bug Detection & Reproduction	[44]	ChatGPT (GPT-3.5-turbo)	App GUI automated testing.	R2,R3,R4, R5,R6 & R9
	[45]	ChatGPT	Generates invalid text inputs to detect bugs.	R4
Malware Detection	[95]	GPT-4	Malware detection	R1,R2,R3,R5, R6,R8 & R9

Chapter 5

MetaLeak

This chapter illustrates how sharing images containing sensitive metadata[1] can lead to the intentional or unintentional leak of users' personal or confidential information. Specifically, we identify two factors contributing to the threat model we focus on in this chapter. Firstly, EXIF metadata is not safeguarded by the permission model or any other protective mechanisms provided by Android. Secondly, the sandbox mechanism is designed to isolate apps in terms of processes and data. However, in reality, not all types of data generated by an app are protected by this mechanism. As outlined in Section 2.1, images stored in public storage are only covered by the permissions model, yet these files contain sensitive metadata affecting user privacy and security. Combining these two factors creates vulnerabilities, which we will discuss below.

In our threat model, we consider risks related to image metadata disclosure from two different perspectives. The first relates to the transparency of personal data collected and shared by apps. As discussed in Section 2.1, Android notifies users through the manifest file of which personal information will be collected and requires user authorization based on the set of app features that users choose to use (runtime permissions). However, EXIF metadata is not collected via APIs; hence, it does not require the user's consent. Therefore, users might be unaware that personal data conveyed through EXIF metadata (e.g., location, time, etc) are collected and shared. Specifically, in our study, we assume that users will only consent to grant three permissions to an app, namely *(READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE)* to access memory and *(INTERNET)* for internet access, which are those sufficient for users to share/send/upload images online. For brevity, we collectively refer to these three permissions as *metadata permissions*. However, we show that the app can collect location, datetime, etc., through EXIF metadata without the end users being explicitly aware. Indeed, if an app unrelated to location tracking or not providing navigation features (e.g., a photo gallery, file manager, photo editor, cloud storage app) requests users to grant location permissions, it may raise suspicion among users. Conversely, asking only for access storage and internet permissions aligns with the app's functionality, making it

[1] To recall, sensitive metadata includes: datetime, smartphone model, smartphone brand, serial number, and GPS.

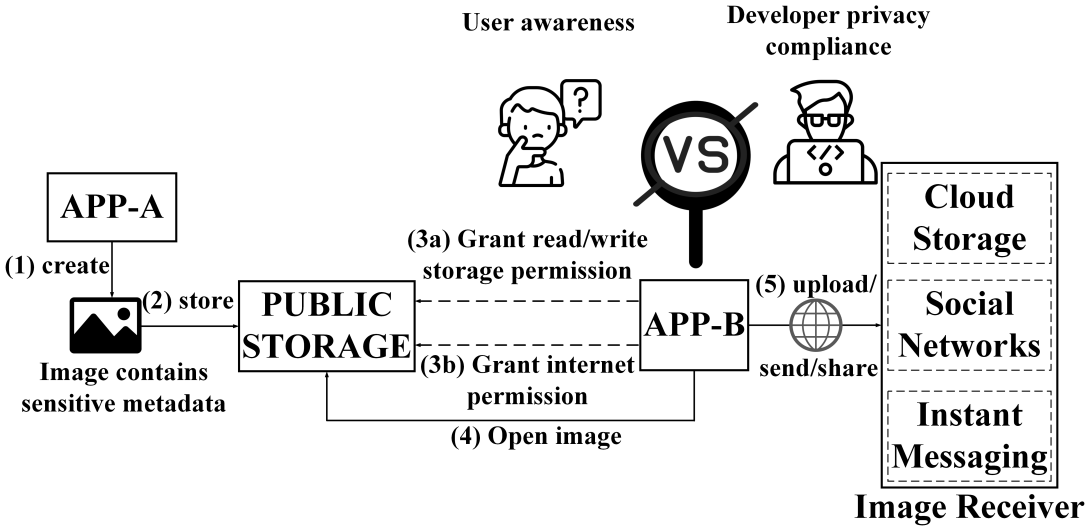

Figure 5.1: Privacy risks of EXIF metadata

more difficult for end users to detect the potential risk. Since EXIF metadata is hidden information, users would find it difficult to imagine being victims of social engineering and re-identification attacks when hackers collect sensitive metadata from their shared images online (cf. Section 3.2).

This threat is illustrated in more detail in Figure 5.1. A user first takes a picture using App-A, which embeds sensitive EXIF metadata into the image (step 1). The image is then stored in the smartphone's public storage, a shared area accessible to all apps (step 2). Later, the user employs another app, App-B, and grants it *metadata permissions* (steps 3a–3b). With these permissions, App-B can read the image from public storage (step 4) and upload it online through cloud storage services, social networks, or messaging apps (step 5).

If App-B transmits the image without removing its EXIF metadata, sensitive information may be leaked. This occurs because App-B does not sanitize metadata prior to transmission and does not notify users that such information will accompany their upload. Although App-B does not generate this metadata, its access to public storage effectively makes this shared location act as a side channel. In our experiments, we evaluated whether App-B removes sensitive metadata before uploading by analyzing the outgoing traffic when processing a test image. Our results show that 21.9% of the tested apps transmitted at least one of the five categories of sensitive metadata (cf. Section 5.2).

To illustrate a typical usage scenario, consider a user taking a photo with the smartphone's pre-installed camera app (App-A), which embeds standard EXIF fields. The image is saved in public storage. The user then uploads it using a cloud storage app

(App-B), for example OneDrive[2], in order to free device space or simplify sharing. After installation, the user grants App-B the required *metadata permissions*. However, our analysis shows that during the upload process, App-B transmits sensitive metadata such as datetime, GPS coordinates, smartphone model, brand, and software kernel (see, for instance, the 8th row of Table 5.1).

The potential consequences are significant. A cloud provider may use datetime and GPS metadata to infer user routines or feed third-party services such as personalized advertisements. Individuals who gain access to shared cloud folders may also determine where and when images were taken, enabling potential tracking. Additional attack scenarios based on model, brand, or serial-number metadata have been documented in prior work (cf. Section 3.2).

The second privacy threat we consider in this chapter is compliance with the developer's privacy policy when collecting personal data through EXIF metadata. We found that many app developers do not consider EXIF metadata when specifying their privacy policies (i.e., Data Safety).

We designed MetaLeak to evaluate the prevalence of threat models and the privacy threat associated with leaking sensitive data through EXIF metadata. MetaLeak adopts a hybrid analysis approach. We first apply static analysis to extract permissions from the app manifest and select apps that request *metadata permissions*. We then perform dynamic analysis by capturing the app's sent-out traffic during image upload, sharing, or sending operations using an MITM proxy. The captured traffic is examined to determine whether sensitive metadata are transmitted. When applying MetaLeak to 5,000 popular apps, we found that 21.9% of the analyzed apps transmitted at least one type of sensitive metadata. Moreover, when focusing specifically on those apps that leaked GPS, our experiments show that only 10.4% of developers comply with their privacy policies, meaning they explicitly declare that their apps collect users' location information.

The remainder of this chapter is organized as follows. Section 5.1 outlines the system architecture and its main components, while Section 5.2 presents the results of our experiments.

5.1 MetaLeak Architecture

When developing MetaLeak, we conduct an in-depth analysis of existing user interaction simulation tools, including Monkey, Appium, and DroidBot. However, these tools all have many weaknesses and require a trade-off between accuracy and scalability (cf. Section 3.1). In addition, these tools do not support setting up a complex testing workflow that involves accessing public storage, selecting an image, and uploading it to a target destination (e.g., Google Drive). Therefore, we develop MetaLeak, leveraging a semi-automated approach to strike a balance between accuracy and scalability across thousands of apps.

In describing Metaleak architecture, we refer to two groups of individuals: **users** and **testers**. Users are individuals exposed to privacy/security risks, whereas testers

[2] https://play.google.com/store/search?q=OneDrive

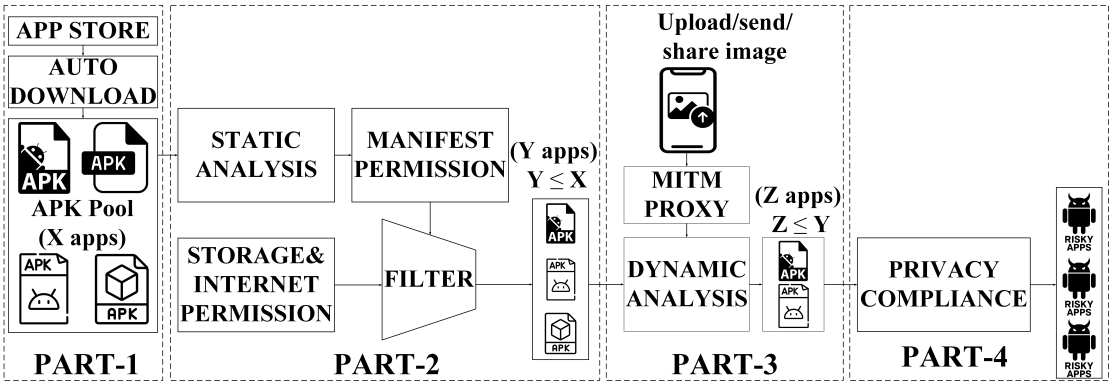

Figure 5.2: MetaLeak Architecture

are the ones who use MetaLeak to assess the vulnerabilities related to EXIF metadata as described in the threat model of this chapter. MetaLeak combines microservice and decentralized architecture, allowing the system to easily upgrade, expand, and enhance its load capability by supporting multiple simultaneous testers. This is a significant advantage, but it requires managing multiple traffic streams from different users simultaneously. Unlike the web, smartphones generate numerous traffic streams from the OS and installed apps. Furthermore, even when these apps are stopped, traffic persists due to Foreground services,[3] etc., continuing to send and receive data. In addition, we also paid attention to cost-effectiveness and convenience for testers. Therefore, we develop a system that can be entirely operated by smartphone simulators instead of requiring real devices, as done by most previous research (e.g., [59, 62]). Smartphone simulators allow testers to investigate apps on various phone models and OS versions.

The architecture of MetaLeak consists of four components, all developed using the Python programming language, as illustrated in Figure 5.2. The first component is devoted to apps downloads (APK files) to form the pool of apps to be examined (part 1). On this pool, we conduct static analysis to retrieve permission requests from the manifest.xml files (part 2). Then, we retain only those apps that request *metadata permissions* as these permissions are the minimum required for the threat model. Subsequently, we perform dynamic analysis on these apps by installing the APKs on a simulator, launching the apps, granting only internet access and read/write storage permissions, selecting images containing sensitive EXIF metadata, and uploading/sending/sharing these images online. We conduct traffic analysis by capturing and decrypting the outbound traffic of apps to examine if the sent-out traffic contains sensitive metadata from the images (part 3). For those apps that transmit sensitive metadata, we crawl the data collection policy, as developers announced in the Data Safety section on the Google Play store, to assess their actual privacy compliance compared to what has been declared (part 4).

In what follows, we describe all the components mentioned above in detail.

[3] https://developer.android.com/guide/components/foreground-services

5.1.1 Apps Download

We use the Androzoo [6] dataset to build our APK files dataset. Androzoo provides information on 23,439,414 apps, including package names that identify unique Android apps on devices, the Google Play Store, and third-party stores. However, as Androzoo lacks essential app metadata (e.g., app name, purpose, number of downloads, and developer), we develop **AutoDownload** based on the Selenium framework.[4] This tool automatically downloads APK files and crawls app metadata by combining the Apkcombo store's database[5] with the Androzoo package names. The output is a set of APKs, and their respective metadata information is stored in a CSV file.

5.1.2 Static Analysis

The primary task of static analysis in MetaLeak is to collect a list of app permissions requested through the manifest file. Then, we conduct filters to keep just those apps requiring internet access and read/write storage permissions. These permissions relate to our threat model. To build MetaLeak with microservices and decentralized architecture, we skip traditional reverse engineering tools like Apktool, Dex2Jar, and Jadx, which require local installation and command-line execution. Instead, we utilize MobSF,[6] an open-source static analysis platform used extensively in previous studies [27]. Leveraging MobSF's REST API, we develop a fully automated static analysis pipeline. This allows testers from anywhere worldwide to perform source code reverse analysis of APK files without additional local installations. Our static analysis service auto-reads the package name of the target apps from the CSV file (output of Section 5.1.1) to select and upload the corresponding APK file to the MobSF server through the REST API endpoint. MobSF server automatically reverse engineers the app to extract the Manifest.xml file and returns it in JSON format. This manifest information is then processed and appended to the CSV file. After static analysis, the APK file is deleted from the MobSF server.

5.1.3 Dynamic Analysis

Dynamic analysis examines apps' outbound traffic when testers upload, share, or send images containing sensitive metadata. Specifically, we store an image with sensitive metadata on the smartphone's public storage. As the tester performs open, upload, send, or share operations through the app, the dynamic analysis module captures and analyzes the app's outbound traffic to identify whether any of the five types of sensitive metadata are transmitted and, if so, which type. Analyzing outbound traffic from Android phones is complex, especially when serving multiple testers simultaneously. To address this, we develop a dynamic analysis pipeline, depicted in Figure 5.3, consisting of a **client** and **server** layer.

[4] https://www.selenium.dev/
[5] https://apkcombo.com/
[6] https://github.com/MobSF/Mobile-Security-Framework-MobSF

At the server layer, we develop an open-source MITM proxy (version 9.0.1)[7] to inspect HTTP/HTTPS methods in outbound traffic from the AVD (i.e., Android Virtual Devices). Specifically, we only capture sent-out traffic using the POST and PUT methods, which are used for image sending/uploading/sharing. The captured traffic is stored as bytes at the server layer. The open-source MITM proxy lacks support for the QUIC (Quick UDP Internet Connections) [38], preventing traffic decryption for some apps utilizing this protocol, a challenge noted by researchers [59]. To address the QUIC Protocol issue, we chose Google Drive as the image-receiving endpoint when testers upload images since it supports this protocol. We create a basic tool based on the Selenium framework to automate image downloads after uploading them to Google Drive, and then check the existence of sensitive metadata. We emphasize that using Google Drive to bypass technical issues We emphasize that using Google Drive to bypass technical limitations related to the QUIC protocol accounts for only a very small portion in practice, as this network protocol is still relatively new and not yet widely adopted. Therefore, to handle the issue comprehensively, we use MITM proxy. This choice is practical because some apps, such as Skype and Snapchat, do not support direct image uploads to Google Drive, and MITM proxy processes traffic faster than retrieving images via Google Drive.

At the client layer, we use the AVD to simulate a device on which to install Android apps (APK files). AVD utilizes the two latest Android OS versions, 12 and 13. To examine HTTPS traffic from the app, we obtain root access and insert the MITM proxy certificate into the Android OS trusted certificate repository using rootAVD, Magisk, and AlwaysTrustUserCerts tools.[8] We develop the Client-service to send ADB (Android Debug Bridge) commands[9] to the AVD, controlling APK installation, traffic redirection to the MITM proxy, and app uninstallation. In some cases, the apps use advanced security measures like SSL pinning [41], which restricts certificate acceptance and prevents MITM interception. We employ Frida[10] to bypass SSL pinning to create hooking API services.

The **Client-service (CS)** and **Server-service (SS)** communicate via Kafka,[11] an open-source distributed event streaming platform for real-time data transmission. Kafka segregates traffic streams from different testers. Each tester joining the dynamic analysis is assigned a unique **tester-ID**, which creates two Kafka topics: **metadata-ID** for sending start/stop commands to the MITM proxy and **bytes-ID** for transmitting captured traffic to the Client-service. For example, a tester with *tester-ID=A* has Kafka topics *"metadata-A"* and *"bytes-A"*. Thus, each tester's traffic is isolated in their Kafka topics. We design our dynamic analysis pipeline to start and stop the MITM proxy for each app under investigation for two main reasons. Firstly, due to the substantial volume of outbound traffic from Android devices, we aim to capture only traffic generated during image upload/share/send, filtering out unrelated traffic like ads or logins. Second, to clear the MITM proxy's cache after each test, ensuring accurate subsequent evaluations.

[7] https://www.mitmproxy.org/
[8] https://github.com/research-mobile-security/MetaLeak
[9] https://developer.android.com/tools/adb
[10] https://frida.re/
[11] https://kafka.apache.org/

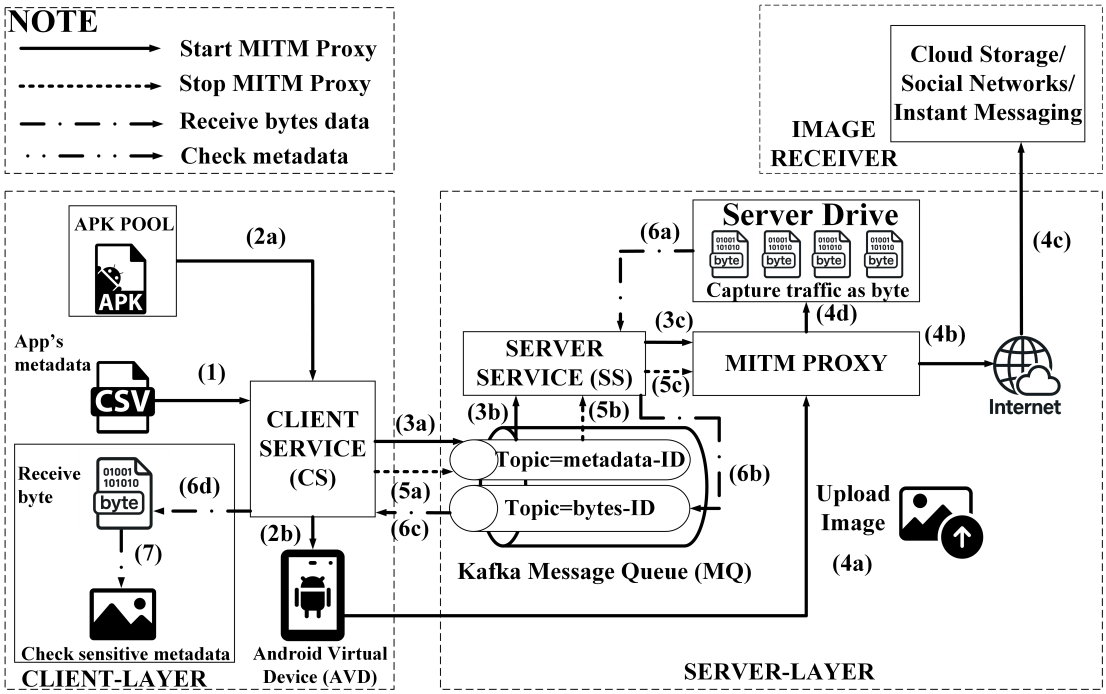

Figure 5.3: Dynamic Analysis

During the upload/sent/share process, images are segmented based on the TCP sliding window size. If a segment is corrupted and triggers a TCP retry, it may be transmitted multiple times, leading to potential false positives in subsequent evaluations.

Algorithm 1 outlines the operations conducted during dynamic analysis, executed by testers to analyze the under-test apps. It takes a set of APK files **(APK_in)** and associated metadata information **(CSV file)** from static analysis as input. The algorithm analyzes the sent-out traffic of apps to identify if they expose sensitive metadata, determining the leak metadata type. The dynamic analysis result is appended to the same CSV file at the corresponding app index to obtain the final result file (cf. column **metadata_leak** of Table 5.1). Additionally, the APK files of apps leaking sensitive metadata are moved to the **APK_out** set.

The algorithm uses a static variable named *Ts-ID* to store the tester-ID value of the considered tester (line 1). The tester has two Kafka topics: *"metadata-Ts-ID"* and *"bytes-Ts-ID"* as explained above. These Kafka topics are used to send commands to trigger the MITM proxy and transmit the captured sent-out traffic. Additionally, the iterator variable - *app_index* (line 2) - represents the position of each app in the input CSV file (row), and it is initialized to 0. The app_index variable is used in a while loop (line 3) and increments until all the apps in APK_in have been tested. Next, the algorithm reads the app's package name from the CSV file (line 4) at the app_index position, selects the corresponding APK file from APK_in, and installs the APK on

Algorithm 1 Dynamic Analysis

Input: APK_in: the set of APK files of target apps and their metadata returned by static analysis, stored as CSV file.

Output: APK_out: the set of APK files of apps leaking image metadata and the type of metadata leakage stored in the input CSV file.

```
1: Ts_ID = "tester-ID"
2: app_index = 0
3: while app_index < length(APK_in) do
4:     packName ← getAPKPackageName(app_index)
5:     installStatus ← installAPKOnAVD(packName)
6:     if installStatus is Success then
7:         runStatus ← runAPKOnAVD(packName)
8:         if runStatus is Success then
9:             startMITM(”metadata-Ts-ID", ”start")
10:            startRedirectTrafficViaMITM()
11:            while uploadImageToInternet() do
12:                bytes ← captureTraffic()
13:            end while
14:            stopRedirectTrafficViaMITM()
15:            stopMITM("metadata-Ts-ID", "stop")
16:            bytesNum ← countBytesCaptured(bytes)
17:            while bytesNum > 0 do
18:                sendBytesToClient("bytes-Ts-ID", bytes)
19:            end while
20:            images ← restoreImage(bytes)
21:            metaDataArr ← checkExistMetaData(images)
22:            if metaDataArr is Null then
23:                CSV_file ← appendCSV(app_index, "no leak of metadata")
24:            else
25:                typeLeak ← checkLeakType(metaDataArr)
26:                CSV_file ← appendCSV(app_index, typeLeak)
27:                APK_out ← moveAPKToOutPool(packName)
28:            end if
29:            deleteAllImage(images)
30:        else
31:            CSV_file ← appendCSV(app_index, "run fail")
32:        end if
33:    else
34:        CSV_file ← appendCSV(app_index, "install fail")
35:    end if
36:    app_index ← app_index + 1
37: end while
38: return CSV_file, APK_out
```

the AVD (line 5). If the app installation process is successful, the algorithm proceeds to the next step; otherwise, the algorithm appends *"install fail"* to the CSV file at the app_index position (line 34) and checks the next app. Upon successful app installation, the algorithm runs the app (line 7). If the app does not run successfully, the algorithm appends *"run fail"* to the CSV file at the app_index position (line 31) and examines the following app. Otherwise, the algorithm sends the *"start"* command on the *"metadata-Ts-ID"* topic (line 9) to the SS to initiate the MITM Proxy, then directs all traffic from the AVD through the MITM Proxy (line 10). Subsequently, the algorithm enters a while loop, waiting for the tester to open and upload the test image online (line 11). This image is prepared and stored in AVD public storage (AVD enables the emulation of public storage akin to physical devices) and contains all five types of sensitive EXIF metadata we focus on. Concurrently, SS captures all outbound app traffic in byte form through the MITM Proxy (line 12). Upon completing the image upload process, CS stops the traffic redirection through the MITM Proxy (line 14) and then sends the *"stop"* command via the *"metadata-Ts-ID"* topic to the SS to terminate the MITM Proxy (line 15). The SS counts the bytes captured (line 16) and transmits all the bytes to the CS on the*"bytes-Ts-ID"* topic (lines 17 – 18). After receiving the whole bytes of traffic, the CS reconstructs them into image files (line 20). Next, the CS checks the EXIF metadata of the images and assigns the results to the *metaDataArr* variable (line 21). If all image files contain no metadata, the algorithm appends *"no leak of metadata"* to the CSV file at the app_index (lines 22 - 23); otherwise, it identifies the types of leaked metadata, since an app might leak more than one (lines 25). Subsequently, the algorithm appends these types of leaked metadata to the CSV file at the app_index position (line 26) and moves the considered APK from APK_in to APK_out (line 27). Next, the algorithm deletes all image files (line 29), increases the *app_index* variable by 1 (line 36), and continues to check the next app. Finally, after all apps have been examined, the algorithm returns the updated CSV_file containing the outcomes of the dynamic analysis of each app. The result is column **metadata_leak** of Table 5.1, and the corresponding APK files for apps leaking metadata are stored in **APK_out** (line 38).

5.1.4 Privacy Compliance Check

Using dynamic analysis, we identify apps that send sensitive metadata and determine exactly which types of metadata appear in their outgoing traffic However, this does not inherently imply that the apps violate user privacy. The app does not breach user privacy if the transmitted metadata matches the developer's disclosures in the app's Data Safety section. Therefore, we compare the leaked metadata from dynamic analysis with the data the app declares to collect. We check the Data Safety section on the Google Play Store to assess compliance. To automate this task, we develop a Privacy Compliance Service (PCS) using the Selenium framework. PCS reads an app's package name from a CSV file obtained after dynamic analysis, accesses the Google Play Store, and crawls information about the data collected by the app. This data is recorded in the CSV file (see column **app_policy** of Table 5.1). We then compare the declared data with the leaked metadata. The app is non-compliant with privacy if sensitive metadata

is transmitted without being listed as collected data. For example, Samsung Email (5th row of Table 5.1) leaks GPS information through image metadata without declaring any data sharing policy, thus violating user privacy.

5.2 Experimental Evaluation

We used MetaLeak to assess the risk of exposing sensitive metadata in various apps. With the AutoDownload module (cf. Section 5.1.1), we download 43,718 apps across multiple categories, such as photography, communication, productivity, etc. We completed the app crawling process in March 2023. Our dataset comprises APK files and the app's metadata stored in a CSV file, including package name, app's name, category, number of installations, developer, app's usage purpose, and download link. Subsequently, we conduct static analysis (cf. Section 5.1.2) on all these apps to collect manifest permissions. Next, we perform filtering to retain only those apps that requested *metadata permissions*, as these permissions are related to our threat model. As a result, we obtain 26,230 apps. Finally, we select the top 5,000 most popular apps based on the number of installations and run the dynamic analysis on these apps. For the 5,000 selected apps, the category distribution is as follows: Photography (79.74%), Beauty & Art (11.4%), Tools (4.52%), Productivity (2.06%), Communication (1.12%), Entertainment (0.36%), Business (0.32%), Personalization (0.30%), Lifestyle (0.10%), and Social (0.08%). Installation statistics are 5% of apps with over 100 million installs, 11.36% between 10 million and 100 million, 22.48% between 1 million and 10 million, 31.08% between 100 thousand and 1 million, and 30.08% with fewer than 100 thousand installs.

The testing team comprises 20 participants: 2 Ph.D. students, 9 master's students, and 9 bachelor's students, all with Computer Science backgrounds from Europe and Asia. Each participant analyzes 250 of the 5,000 apps. Dynamic analysis averages 3 minutes per app, with an additional 1 hour for AVD setup, totaling approximately 12.5 hours per tester. For testing, we deploy MetaLeak's Server-layer on Amazon EC2. We distribute APK files, MITM proxy certificates, and a single image containing sensitive metadata to testers. We emphasize that testers only utilize one image during the testing process. The values of the five sensitive metadata types were fixed as a static array in the source code, allowing the Client-service to compare the presence of metadata in captured sent-out traffic against these values. Testers were instructed to install the AVD, obtain root access, and set up the MITM proxy certificate in the Android OS trusted credentials. We provide each tester with a *tester-ID* (for Kafka topics) and *port number* (for microservice architecture) as described in Section 5.1.3. Testers only need to update their respective tester-ID and port number in the Client-service source code. Testers were also requested to create a directory on their Google Drive as the destination for the upload process. When testers execute the Client-service, this service automates the selection, installation, and execution of the target app's APK. Once the app runs successfully, the Client-service starts the MITM proxy, redirects traffic, and awaits tester interaction with the AVD. Testers only need to grant read/write storage and internet access permissions and then upload the image to Google Drive. The MITM

Table 5.1: MetaLeak's analysis results, sorted by installation number (B: billion, M: million).

No	App name	Package name	Metadata leak	Installs	Purpose	App policy
1	Gmail	com.google.android.gm	datetime (D), GPS, model (M), brand (B), software (SW)	10B	Message-Mail	Location (LOC), Personal (PER), Financial (FIN), Message (MES), Photo (PHO), Video (VID), Audio (AUD), File (FIL), Calendar (CAL), Contact (CON), App performance (APE), Device ID (DID)
2	Google Drive	com.google.android.apps.docs	D, M, B, SW	5B	Cloud storage	LOC, PER, FIN, MES, PHO, VID, AUD, FIL, CAL, CON, APE, DID
3	Google Photos	com.google.android.apps.photos	D, GPS, M, B, SW	5B	Cloud storage	LOC, PER, FIN, MES, PHO, VID, AUD, FIL, CAL, CON, APE, DID
4	Snapchat	com.snapchat.android	D, M, B, SW	1B	Message-Mail	LOC, PER, FIN, MES, PHO, VID, AUD, CON, Web browsing (WEB), APE, DID
5	Samsung Email	com.samsung.android.email.provider	D, GPS, M, B, SW	1B	Message-Mail	Not specified for the collected data
6	Xiaomi File	com.mi.android.globalFileexplorer	D, GPS, M, B, SW	1B	File manager	Not specified for the collected data
7	Samsung Files	com.sec.android.app.myfiles	D, M, B, SW	1B	File manager	Not specified for the collected data
8	OneDrive	com.microsoft.skydrive	D, GPS, M, B, SW	1B	Cloud storage	PER, PHO, VID, APE
9	Dropbox	com.dropbox.android	D, GPS, M, B, SW	1B	Cloud storage	PER, FIN, PHO, VID, AUD, FIL, CON, APE, DID
10	Files by Google	com.google.android.apps.nbu.files	D, GPS, M, B, SW	1B	Cloud storage	PER, MES, APE, DID

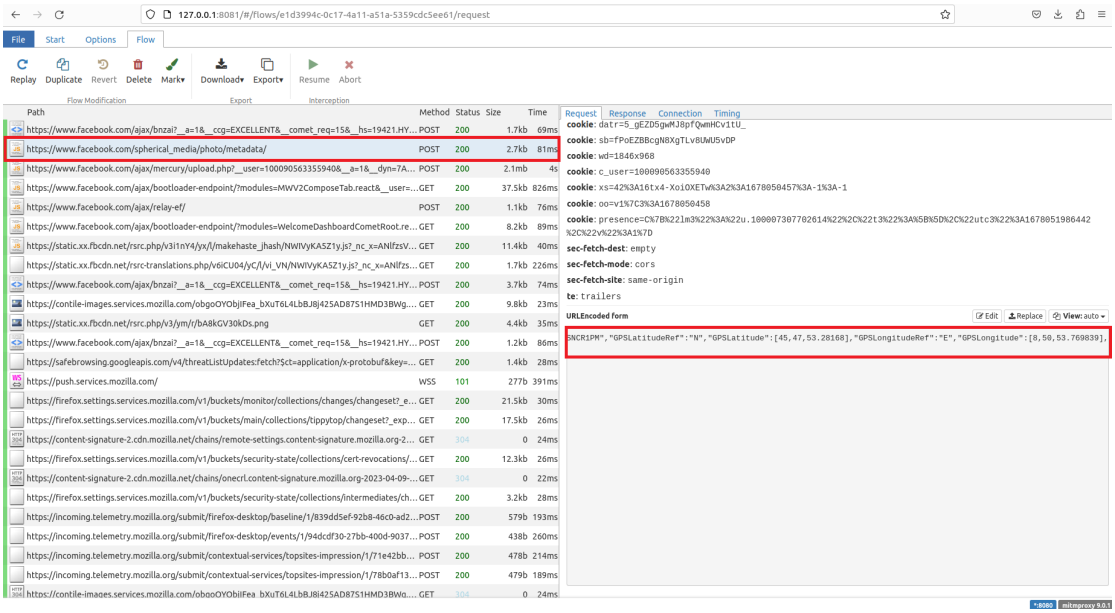

Figure 5.4: GPS detected in the sent-out traffic of the Facebook app.

proxy captures traffic in parallel with the image upload. After the upload, the Client-service stops the MITM proxy, collects the captured traffic, decrypts it, and checks for the presence of sensitive metadata. If the MITM proxy fails to capture the traffic, the Client-service triggers a script to download the image from Google Drive and check its metadata, as detailed in Section 5.1.3.

Figure 5.4 shows the sent-out traffic captured by the MITM proxy for the Facebook app. When the tester uploads an image through the app, the proxy indicates that the GPS metadata embedded in the image is extracted and transmitted to Facebook's servers via the endpoint https://www.facebook.com/spherical_media/photo/metadata/. This indicates that Facebook actively collects sensitive metadata from photographs that users post, rather than removing it.

The results for our dataset of 5,000 apps are presented in Figure 5.5. The pie chart includes four labels: **(1) Install fail (18.7%), (2) Function fail (9.6%), (3) No leak of sensitive metadata (49,8%)**, and **(4) Leak of sensitive metadata (21.9%)**. The **"Install fail"** label represents apps that failed to install on the device due to incompatibility with the latest Android OS, such as libraries not being updated, apps unable to connect to service provider's servers or API gateways, etc., leading to the app crashing and stopping immediately. Note that the successful installation of an app can only be determined when the app is actually installed on an AVD or a physical device and not through the static analysis of the app's code. The **"Function Fail"** label signifies apps that were installed but did not run successfully or failed to execute the specific steps of our testing scenario, including opening and uploading/sharing/sending images online. The **"No leak of sensitive metadata"** label represents apps that do not

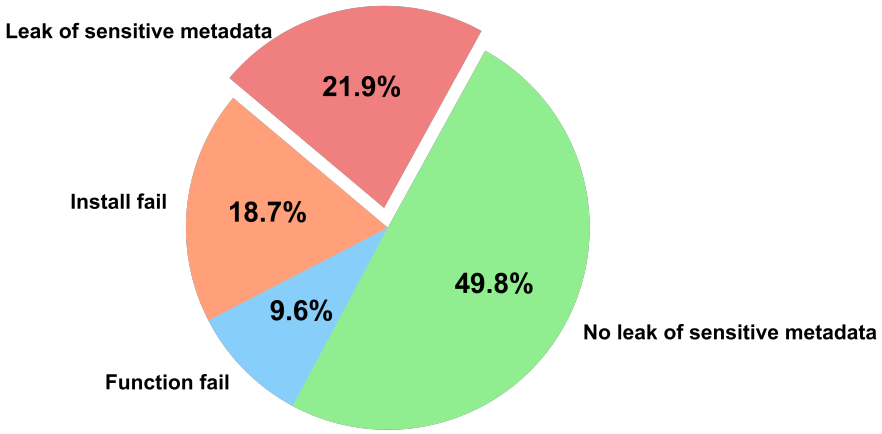

Figure 5.5: Results for the considered 5,000 apps

contain sensitive metadata in their sent-out traffic. In contrast, the **"Leak of sensitive metadata"** label describes apps with at least one of the five types of sensitive metadata we consider in their sent-out traffic. Figure 5.5 shows that 21.9% of the 5,000 analyzed apps **(≈ 1095 apps)** leaked at least one type of considered sensitive metadata. This means that the risk we consider in our threat model is indeed relevant. By comparing our findings with Google's 14 sensitive data categories and their corresponding data types in Table 2.1, we observe that 4/5 sensitive metadata types we analyze, namely, datetime, smartphone model, smartphone brand, and serial number, are not included in Google's official documentation, despite prior research demonstrating that these data can be exploited to compromise user privacy (cf. Section 3.2). This omission creates a legal gray area regarding GDPR/CCPA compliance. Therefore, although 21.9% of apps that leak at least one type of sensitive metadata cannot be definitively classified as violating privacy regulations due to Google's unclear legal framework, it is evident that these apps are at a significantly higher risk of being exploited by attacks targeting sensitive metadata. For GPS, the only sensitive metadata type that Google considers sensitive data, as explained in Section 5.1.4, the presence of this data in sent-out traffic does not imply that the app is violating privacy if the developer has clearly stated that the app will collect GPS in the Data Safety section. Therefore, we focus on evaluating the app's privacy compliance with respect to GPS, and the corresponding results are presented in the following part of this section.

We conduct an in-depth analysis to identify the most commonly leaked types of sensitive metadata. It is important to emphasize that an app may leak one, several, or all five types of metadata (refer to the **metadata_leak** column of Table 5.1 for examples). The statistical results are as follows: 680 apps leaked GPS information, 1043 apps leaked datetime information, 1055 apps leaked smartphone model information, 1055 apps leaked smartphone brand information, and 998 apps leaked smartphone software kernel information. These results indicate a relatively balanced leakage rate across all

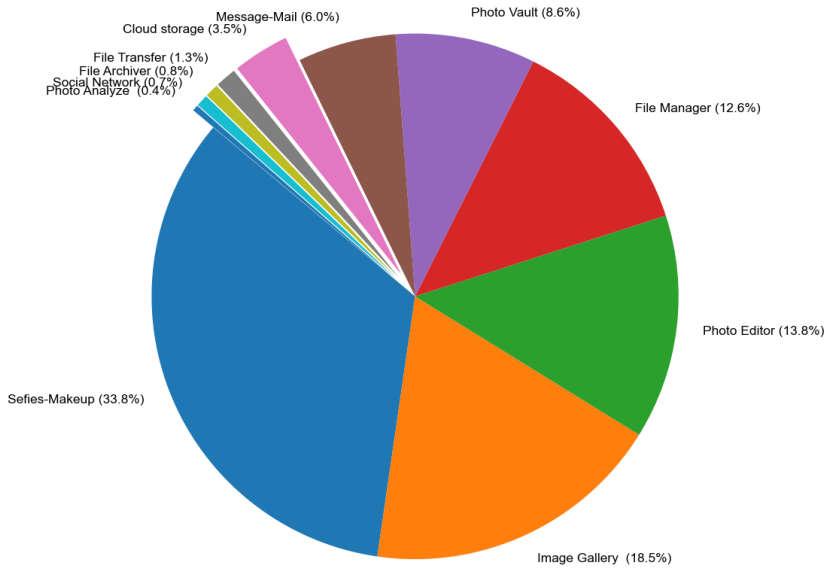

Figure 5.6: Usage purpose of apps leaking sensitive metadata

five metadata types. Notably, 97.81% of the 1095 apps leaking sensitive metadata leaked almost all types (four or five), while only 2.19% (24 apps) leaked just one type.

Furthermore, we examine the **usage purpose** of those apps that leaked sensitive metadata. The results show that out of 1095 apps leaking sensitive metadata, the predominant purpose categories include Selfies-Makeup (33.8%), Image Gallery (18.5%), Photo Editor (13.8%), File Manager (12.6%), and Photo Vault (8.6%). The remaining purpose categories, which are smaller proportions, include Message-Mail (6%), Cloud Storage (3.5%), File Transfer (1.3%), File Archiver (0.8%), Social Network (0.7%), and Photo Analyzer (0.4%) (cf. Figure 5.6). Although cloud storage and message-mail apps have a lower proportion, Table 5.1 (column **installations**) reveals that these apps have many installations, reaching billions of users and thus resulting in very high risks. For instance, the message-mail app provided by Google (the 1st row of Table 5.1) exhibits the highest installation count, reaching up to 10 billion installations. The app's purpose analysis results confirm that leakage of information through EXIF metadata is challenging to detect by end users since it is mainly due to apps that do not provide primary functionalities related to leaked information (e.g., location tracking, navigation, calendar management, accessing OS kernel information, etc).

Next, we present the results of the privacy compliance assessment for apps leaking GPS metadata (680 apps). Specifically, we collect information on the Google Play Store on the type of data that developers specify will be collected by those apps to see if they mention GPS data. Our experiments found that only 10.4% of 680 apps leaking GPS metadata ($\approx$ 71 apps) declared they would collect users' location information. This means that the remaining portion, 89.6% ($\approx$ 609 apps), collects GPS information through image metadata when the users upload/share/send the image online, but they

do not declare it in their data safety policy. These apps not only violate Google's data safety policy but also fail to adhere to the *transparency* principles of GDPR/CCPA. Specifically, among those 609 non-compliant apps, 68% (≈ 414 apps) claim not to collect the user's location but to gather other information (e.g., messages, contacts, finances, etc.), and 29% (≈ 177 apps) state that they do not collect user data. That means developers explicitly state that the app does not collect any data, but in fact, it gathers GPS through EXIF metadata. Moreover, 3% (≈ 18 apps) do not provide information about the data type they collect.

Table 5.1 provides an in-depth analysis of the findings of our experimental results for the top 10 apps in terms of the number of installations. In the table, the column **app_name** represents the application name, column **pkg_name** is the unique identifier of the app on the app store, column **metadata_leak** represents the outcome of dynamic analysis, column **installations** reports the number of installations (where B denotes billion and M denotes million), whereas the column **app_purpose** denotes the usage purpose of the app. Finally, the column **app_policy** records the type of user data collected by the apps (Data Safety), as stated in Section 5.1.3. For example, the app *Xiaomi File* (with one billion installations) states that it will not collect any user data *(not specified for the collected data)* but leaks GPS information. Similarly, the app *Files by Google* (with one billion installations) does not include GPS in the list of collected data, yet it leaks GPS location through image metadata. Therefore, these apps do not comply with the declared privacy policies.

Moreover, for 1095 apps identified as leaking sensitive metadata, we find there are three approaches for handling EXIF metadata: (1) using Google's built-in class *android.media.ExifInterface* (old version),[12] (2) using Google's built-in class *androidx.exifinterface.media* (new version),[13] and (3) using self-developed functions. Among these apps, 596 use *android.media.ExifInterface*, 223 use *androidx.exifinterface.media*, and 276 employ self-developed functions. We review Google's documentation for two built-in classes *android.media.ExifInterface* and *androidx.exifinterface.media*. Google highlights issues with the old version and recommends that developers use the new version. Despite the newer class offering more methods for handling EXIF metadata (28 compared to 21 in the old version), neither class provides methods for partially or completely removing EXIF metadata. The absence of methods for removing EXIF metadata in the Android class may contribute to the security risks discussed in this chapter. Even when developers are aware of the risk, they lack the practical tools provided by the Android OS. This highlights that not only do Google's permission model and sandbox contain exploitable vulnerabilities functioning as side-channels, but also that Google underestimates the risk of sensitive metadata leaks through the online upload, sharing, or sending of images.

Moreover, because Google does not classify datetime, smartphone model, smartphone brand, or serial number as sensitive data and does not provide any corresponding manifest permissions to protect them, this creates a legal grey area and makes it difficult to

[12] https://developer.android.com/reference/android/media/ExifInterface
[13] https://developer.android.com/reference/androidx/exifinterface/media/ExifInterface

determine responsibility when attackers exploit these four types of sensitive metadata. Therefore, for these four data types, we consider that developers are not intentionally violating user privacy. In contrast, GPS metadata is explicitly recognized by Google as sensitive data. Hence, for all apps that transmit GPS information along with images in their sent-out traffic, we consider this behavior intentional.

Finally, the findings on how apps handle EXIF metadata, together with the root cause (i.e., the lack of EXIF-removing mechanisms), form the basis for our development of ALIBIS in the next chapter.

Chapter 6

ALIBIS

In Chapter 5, we present MetaLeak [52], a semi-automated framework based on hybrid analysis to detect the presence of sensitive metadata in the app's sent-out traffic when users share images online. The advantage of MetaLeak is that it allows us to collect irrefutable evidence of privacy violations through observing the app's outgoing traffic. However, since MetaLeak relies on dynamic analysis, its biggest disadvantage is that it requires human participation to test the app. Specifically, in the dynamic analysis step of MetaLeak, the tester must repeatedly select an image from public storage (e.g., an SD card), use the built-in image-sharing function on the app under investigation, and send the image to the recipient (e.g., cloud storage or a social network). This is a tedious and time-consuming task. Specifically, each tester takes 3 minutes to test an app. The semi-automatic dynamic analysis makes MetaLeak unsuitable for large-scale assessment of sensitive metadata leakage risks. Additionally, observing the app's sent-out traffic in the form of unencrypted data, MetaLeak requires the tester to root their phone and install the SSL certificate of the man-in-the-middle proxy used by MetaLeak. This is a challenging task, and there is no standard implementation method, as rooting the phone is not recommended by smartphone manufacturers.

To overcome these limits, in this chapter, we present **ALIBIS**[1](Assessing and mitigating the risk of sensitive metadata Leakage In moBile Image Sharing), an automated framework to estimate the risk of sensitive metadata leakage. The purpose of ALIBIS is to analyze an app's source code to determine how EXIF metadata is handled. In particular, ALIBIS analyzes only code blocks related to EXIF metadata to determine their specific handling of each type of metadata (e.g., retaining GPS information, deleting datetime information, or removing all metadata). We refer to this step as **code summarization**. In particular, ALIBIS makes use of two approaches for code summarization depending on how the app processes EXIF metadata, one for the apps utilizing Google's built-in classes (i.e., **Pre-provided Lib Group**) and the other for those apps using self-developed functions (i.e., **Self-Developed Group**). For apps utilizing Google's built-in classes, we exploit the method descriptions in Google's documentation to understand how apps handle EXIF metadata (cf. Section 6.2.1). However, Google's

[1] https://github.com/research-mobile-security/ALIBIS

built-in classes do not provide direct methods for handling three of the EXIF metadata, namely *camera brand*, *camera model*, and *device serial number*, which can only be handled through generic methods, such as *getAttribute()* and *setAttribute()*. Therefore, code summarization for code blocks handling these metadata types is performed using code similarity techniques.

In contrast, we cannot rely on Google documents or code similarity for code summarization of blocks exploiting self-developed functions. To handle these blocks, we leverage Large Language Models (LLM) (cf. Section 6.2.2). LLM are powerful deep learning models pre-trained on extensive datasets, which have been proven effective in code summarization tasks [3]. Although LLM offer superior capabilities in understanding code (cf. Section 2.3), they still have certain limitations. Indeed, while LLM can understand code blocks related to EXIF metadata, their ability to generalize the source code's purpose and detail the processing steps for each type of sensitive metadata is limited if the context is not provided. To address this, we propose integrating Retrieval-Augmented Generation (RAG) (cf. Section 2.3.3.2) and LLM. This combination provides an appropriate context to enhance LLM accuracy compared to previous studies that primarily relied on few-shot learning [3,4] or natural language processing (NLP) [7]. Indeed, previous approaches could only predict the purpose of a code block based on function or method names without discerning the specific steps performed by the code block.

Thanks to code summarization, ALIBIS aims to estimate the risk of leaking sensitive metadata by understanding how developers handle image metadata during app development. ALIBIS's main benefit is delivering early warnings, rather than simply detecting evidence of data leakage after it has occurred, as MetaLeak does. Specifically, ALIBIS aims to be integrated easily into existing app analysis systems, such as automated analysis systems for app marketplaces like the Google Play Store. ALIBIS enables app store operators to automatically estimate apps at risk of leaking sensitive metadata via the app's source code. This capability may allow app store operators to deny developers from releasing apps on the app store, reducing user exposure to potentially dangerous apps. Additionally, thanks to MLaaS architecture, ALIBIS can be deployed both on-premise and cloud-based, enabling it to serve multiple app stores simultaneously.

Besides developing ALIBIS, we offer a mitigation solution to prevent the risk of sensitive metadata leakage. We release **ExifMetadataLib**,[2], an Android library that helps users control sensitive image metadata when sharing online. Specifically, when integrated into apps, ExifMetadataLib alerts users about EXIF metadata in their images and enables them to delete some or all of the sensitive metadata.

The remainder of this chapter is organized as follows. Section 6.1 presents the general workflow and concepts we used to develop ALIBIS, while Section 6.2 highlights the methodology behind ALIBIS's design. Section 6.3 details ALIBIS's architecture and implementation. Finally, Section 6.4 presents the results of our experiments and the solutions we propose for mitigating the risk of sensitive metadata leakage.

[2] https://github.com/research-mobile-security/ExifMetadataLib

Table 6.1: Code2vec Example

No.	*Code-1 (A_i, target code block)*	*Code-2 (B_j, labeled code block)*

1	`import java.util.HashMap;`
	`import java.util.Map;`
	`public class CameraUtility {`
	`  public static Map<String, String> addCameraInfo(`
	`  Map<String, String> exifArray,`
	`  String cameraModel,`
	`  String focalLength) {`
	`  exifArray.put("Model", cameraModel);`
	`  exifArray.put("FocalLength", focalLength);`
	`  exifArray.put("WhiteBalance", "Auto");`
	`  exifArray.put("Flash", "Off");`
	`  return exifArray;`
	`  }`
	`  public static void main(String[] args) {`
	`  Map<String, String> exifData = new HashMap<>();`
	`  exifData = addCameraInfo(exifData,`
	`  "Canon EOS R5", "50mm", "f/1.8");`
	`  System.out.println(exifData);`
	`  }`
	`}`
		`import java.util.HashMap;`
		`import java.util.Map;`
		
		`public class ExifUtility {`
		`  public static Map<String, String>`
		`  setExifAttribute(`
		`  Map<String, String> exif,`
		`  String cameraModel,`
		`  String cameraMake,`
		`  String lensModel) {`
		`  exif.put("Model", cameraModel);`
		`  exif.put("LensModel", lensModel);`
		`  exif.put("ISO", "100");`
		`  exif.put("ExposureTime", "1/250");`
		`  return exif;`
		`  }`
		`  public static void main(String[] args) {`
		`  Map<String, String> exifData = new HashMap<>();`
		`  exifData = setExifAttribute(exifData,`
		`  "Nikon D850", "Nikon", "Nikkor 24-70mm");`
		`  System.out.println(exifData);`
		`  }`
		`}`

2	*AST of Code-1*	*AST of Code-2*

3	`MethodDeclaration(name='addCameraInfo')`
	`  -> Call(put)`
	`  -> args=['Model', cameraModel]`
	`  -> Call(put)`
	`  -> args=['FocalLength', focalLength]`
	`  -> Call(put)`
	`  -> args=['WhiteBalance', 'Auto']`
	`  -> Call(put)`
	`  -> args=['Flash', 'Off']`
	`  -> Return(exifArray)`
	
	`MethodDeclaration(name='main')`
	`  -> LocalVariableDeclaration(name='exifData',`
	`  type='HashMap')`
	`  -> Call(addCameraInfo)`
	`  -> args=[exifData, 'Canon EOS R5',`
	`  '50mm', 'f/1.8']`
	`  -> Call(println)`
	`  -> args=[exifData]`
		`MethodDeclaration(name='setExifAttribute')`
		`  -> Call(put)`
		`  -> args=['Model', cameraModel]`
		`  -> Call(put)`
		`  -> args=['LensModel', lensModel]`
		`  -> Call(put)`
		`  -> args=['ISO', '100']`
		`  -> Call(put)`
		`  -> args=['ExposureTime', '1/250']`
		`  -> Return(exif)`
		
		`MethodDeclaration(name='main')`
		`  -> LocalVariableDeclaration(name='exifData',`
		`  type='HashMap')`
		`  -> Call(setExifAttribute)`
		`  -> args=[exifData, 'Nikon D850',`
		`  'Nikon', 'Nikkor 24-70mm']`
		`  -> Call(println)`
		`  -> args=[exifData]`

4	*Embedding Vectors of Code-1*	*Embedding Vectors of Code-2*

5	`[[ 0.0001995 ... -0.0053115 ]`
	`  ...`
	`  [ 0.00480354 ... -0.00157119]]`
		`[[ 0.0001349 ... -0.00532909]`
		`  ...`
		`  [ 0.00474 ... -0.0016004 ]]`

6.1 ALIBIS Workflow

Starting from the APK of the target app to be analyzed, ALIBIS undergoes four steps. The first step is the APK filtering step (cf. Section 6.3.1), where we inspect the APK's Manifest.xml file to determine if the app has requested all **metadata permissions** (i.e., read/write storage and internet permission), required to allow accessing, uploading, sending, or sharing images. Specifically, we use READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, and INTERNET keywords for screening APK files. That is, we only keep apps whose manifests list the three permissions

mentioned above.[3] Next, we decompile the target app to extract its source code, which includes both Java code and files related to the user interface (such as XML files) (cf. Section 6.3.2). Following this, we analyze the source code and extract only code snippets that handle EXIF metadata, called *EXIF-related code blocks* (cf. Section 6.3.3). In the final stage, we conduct code summarization on the extracted code blocks to determine how the app handles EXIF metadata, using the code summarization module (cf. Section 6.3.4). In Section 6.3, we present the implementation details for each phase. Hereafter, we focus on the ALIBIS core element, that is, code summarization strategies.

To better design a proper code summarization strategy, we first analyze 5,000 apps used in the MetaLeak project [52] to determine *how they handle EXIF metadata*. Through this analysis, we identify three methods Android apps use to process EXIF metadata: **(1) method-1**: using *android.media.ExifInterface*[4] (i.e., 40.28% of the apps in the MetaLeak dataset), **(2) method-2**: utilizing *androidx.exifinterface.media*[5] (i.e., 21.20% of the apps in the MetaLeak dataset), and **(3) method-3**: employing *self-developed functions* (i.e., 38.52% of the apps in the MetaLeak dataset). Thanks to this analysis, we observe that each app employs only one of the three methods above, without combining them simultaneously.

Method-1 and 2 both leverage Google's built-in classes, where method-2 was introduced on December 13, 2023, as an upgrade over method-1 to fix an incompatibility issue with the new Android OS. The primary difference between method-1 and method-2 is the number of EXIF functions: 21 for method-1 and 28 for method-2. Therefore, apps that use method-1 or 2 can be classified into a single group, i.e., denoted as the **Pre-provided Lib Group**, representing 61.48% of the apps in the MetaLeak dataset. In contrast, apps that use method-3 are denoted as the **Self-Developed Group** and represent 38.52% of the apps in the MetaLeak dataset.

Therefore, we propose two different code summarization techniques for the two groups. For those apps in the Pre-provided Lib Group, we exploit Google documentation and code similarity to determine how sensitive metadata are processed (cf. Section 6.2.1). Conversely, for apps in the Self-Developed Group, we leverage LLM and improve model accuracy using RAG and a summary chain (cf. Section 6.2.2).

6.2 Code Summarization Methodology

Given a target app $\mathcal{A}$, ALIBIS generates a code summarization for each of its EXIF-related code blocks, denoted as $A_i, i \in [1, n]$. This is done using two distinct approaches, one for apps in the Pre-provided Lib Group and one for the Self-Developed Group. In

[3] It is relevant to note that starting with Android version 13, the READ_EXTERNAL_STORAGE permission is split into three separate permissions to specify the type of data: READ_MEDIA_IMAGES (used to access images), READ_MEDIA_VIDEO (used to access videos), and READ_MEDIA_AUDIO (used to access audio). As the apps considered in the dataset only support Android version 12, we do not include the READ_MEDIA_IMAGES permission in the keywords, although it can be easily integrated in future releases of the framework.

[4] https://developer.android.com/reference/android/media/ExifInterface

[5] https://developer.android.com/reference/androidx/exifinterface/media/ExifInterface

both cases, for each EXIF-related code block A_i, the code summarization step returns one or more labels in the format $\langle$*metadata type*, *shared/removed*$\rangle$, indicating whether the code block shares/removes that specific metadata. Note that the strategies we develop for both the two groups rely on the existence of a knowledge base. As discussed in the following, the knowledge base is exploited for checking code similarity for those apps in the Pre-provided Lib Group, whereas it is used as an external data source for RAG for those apps in the Self-Developed Group. For both purposes, as a knowledge base, we utilize the apps analyzed by MetaLeak. Specifically, in Chapter 5 [52], we apply MetaLeak to assess the risk of sensitive metadata leakage in 5,000 popular apps (see Section 6.4.1 for more details on data validation). The MetaLeak results show that 21.9% ($\approx$ 1095 leaking apps) leaked at least one of the five categories of sensitive metadata. Additionally, MetaLeak results also label all leaking apps (aka their code blocks) with the specific type of sensitive metadata they leaked (e.g., GPS, datetime). Since there is a large difference in the number of leaking and non-leaking apps, to avoid an imbalanced dataset for ALIBIS that could cause classification bias [25], we do not use the entire MetaLeak dataset of 5,000 apps. Instead, we select a subset of the MetaLeak dataset, which we refer to as Labeled Apps Dataset - LAD, where with LAD_{lib_kb} (LAD_{self_kb}) we denote apps belonging to the Pre-provided Lib Group (Self-Developed Group) used as knowledge base in the corresponding code summarization approach (see Section 6.4.1 for more details).

6.2.1 Code summarization for the Pre-provided Lib Group

It is relevant to note that Google provides *direct methods* to handle only *datetime* and *GPS* metadata (e.g., *getDateTime()*, *getLatLong()*, *setLatLong()*, and *setGpsInfo()*), whereas *camera brand*, *camera model*, and *device serial number* can be handled through *indirect generic methods*, such as *getAttribute()* and *setAttribute()*.

In case the code block A_i uses direct methods, we can exploit Google's documentation, in particular the method description, to determine how A_i processes the datetime and GPS metadata. As an example, if a code block uses the direct method *setLatLong(latitude,longitude)*[6] whose description is *"store GPS"*, we can determine that the code block retains GPS data during EXIF metadata processing. This allows us to associate with the code block, aka its corresponding app, the label $\langle GPS, shared\rangle$. For this purpose, we analyze each method's description and associate a label with the corresponding direct methods. More formally, we denote with $\mathcal{D}$ the set of pairs $\langle$method_name, label$\rangle$ containing the method name and corresponding label. Then, we defined a mapping function (f_{mapping}) to associate with each EXIF-related code block A_i, exploiting a direct method, the corresponding label.

We cannot apply the mapping function (f_{mapping}) for code blocks handling EXIF metadata with indirect methods (i.e., *camera brand*, *camera model*, and *device serial number*), as the method description is too generic. To handle these code blocks, we

[6] https://developer.android.com/reference/androidx/exifinterface/media/ExifInterface#setLatLong(double,double)

exploit the *code2vec* model[7] to measure code block similarity. The key idea is to compare the code block A_i with code blocks of apps in the knowledge base. The principle is that if the two blocks have similar code, they handle in a similar way the EXIF metadata, thus they can share the same label. Thus, if A_i is similar to any one of the available labeled blocks, A_i is labeled accordingly. For the *code2vec* code similarity, we exploit apps in LAD_{lib_kb}.

According to the *code2vec* model, we utilize the Abstract Syntax Trees (AST) to represent the syntactic structure of a code block abstractly. AST removes unnecessary details (such as formatting, whitespace, or specific syntax) and focuses on capturing the higher-level structural relationships within the code block. The tree nodes correspond to the source code's components, such as function declarations, loops, conditional expressions, and operations. We generate the AST for the target code block A_i, as well as for code blocks of apps in LAD_{lib_kb}. Note that labels in the MetaLeak dataset cover all five categories of EXIF sensitive metadata. However, since the code block similarity is exploited only for blocks using indirect methods, we generate ASTs only for those code blocks of apps in LAD_{lib_kb} with labels referring to camera brand, camera model, and serial number. We refer to this set of code blocks as CB_{lib_kb}.

Next, all generated ASTs are converted into embedding vectors (i.e., numeric vectors). In what follows, given a code block B_i, we denote with v_{B_i} the corresponding embedding vector.

Thus, to determine the label for A_i, we compute the cosine similarity between the embedding vector v_{A_i}, and the embedding vector v_{B_j}, for each B_j in CB_{lib_kb}, as follows:

$$cosine\_similarity\left(v_{B_j}, v_{A_i}\right) = \frac{v_{B_j} \cdot v_{A_i}}{\left\|v_{B_j}\right\| \left\|v_{A_i}\right\|}$$

Where $v_{B_j} \cdot v_{A_i}$ is the dot product of the two vectors v_{B_j} and v_{A_i}, and $\|v\|$ denotes the norm of vector v. Cosine similarity ranges between -1 and 1, where 1 indicates a high similarity between the two code blocks.

We consider two code blocks to be similar if their cosine similarity is greater than or equal to 0.85. [8] Thus, to determine the label for A_i, we compute its cosine similarity with each code block B_j in CB_{lib_kb}. Next, we use a cosine similarity threshold of 0.85 to select

[7] Code2vec [7] is a neural model that embeds code snippets into fixed-length vectors (i.e., embedding vectors), enabling source code to be processed similarly to natural language. The code2vec constructs an Abstract Syntax Tree (AST) from a code snippet and its label (e.g., method name) and synthesizes these paths into a single vector. This vector facilitates tasks such as method name prediction, identifying similar code snippets, and semantic code classification. However, code2vec relies on a closed-label space. It can only predict labels observed during training. Therefore, the model requires substantial training data and computational resources due to sparsity label issues, which can lead to struggles with untrained labels. While code2vec can summarize code, its method name-based predictions limit its use for ALIBIS. Specifically, it can not determine the exact operations a code snippet performs on a data variable. Nonetheless, we incorporated the idea of using code embeddings to identify similar code snippets in ALIBIS. https://github.com/tech-srl/code2vec

[8] We select a cosine similarity threshold of 0.85 because it is widely used in prior work to indicate near-identical embeddings [5,89], while offering a balanced trade-off between avoiding false positives (i.e., overly loose matching) and minimizing false negatives (i.e., missing genuinely highly similar items).

a set of similar code blocks, i.e., $S_{B_j} = \{B_j \mid 0.85 \leq \text{cosine\_similarity}(v_{A_i}, v_{B_j}) \leq 1\}$. Then, we choose the top 5 code blocks with the highest cosine similarity from S_{B_j} and assign their labels to A_i. Note that A_i is assigned a combined label of the similar code blocks B_j's label. For example, if B_1 is labeled as *share camera brand* and B_2 is labeled as *share camera brand and camera model*, then A_i, which is similar to both B_1 and B_2, will be assigned the label *share camera brand and camera model*.

Example 1. *Let us assume that A_i is the code block denoted as Code-1 in row 1 of Table 6.1. Moreover, suppose that we want to compare it with the code block $B_j \in CB_{lib\_kb}$ whose code is denoted as Code-2, in row 1 of Table 6.1. The ASTs generated from these code blocks, AST of Code-1 and Code-2, respectively, are shown in row 2 of Table 6.1. Subsequently, we transform both ASTs into embedding vectors (i.e., v_{Code-1} and v_{Code-2}), as shown in row 3 of Table 6.1. Finally, we compute cosine similarity of v_{Code-1} and v_{Code-2}:*

$$cosine\_similarity\,(v_{Code-1}, v_{Code-2}) = 0.9961465$$

Since the resulting value is greater than 0.85, we consider blocks A_i and B_j similar, and if B_j is among the top 5 most similar code blocks, we associate B_j's label to A_i.

6.2.2 Code summarization for Self-Developed Group

To achieve optimal performance on specific applications, LLM often require fine-tuning on domain-specific data. For this purpose, we evaluate two prominent techniques: Retrieval-Augmented Generation (RAG) and Few-Shot learning (FSL). In the next section, we explain the reasoning behind our selection of RAG. We then present the proposed framework that combines RAG and LLM.

6.2.2.1 RAG and FSL for EXIF code summarization

As discussed in Section 2.3.2, training or fine-tuning a large language model for specific tasks is theoretically feasible, but it faces two significant challenges: the need for expensive hardware and the difficulty of creating a supervised dataset for specialized software engineering tasks, which is often impractical for most organizations [4]. Therefore, instead of training or fine-tuning LLM specifically for ALIBIS, we leverage a pre-trained LLM and optimize input contexts to enhance LLM's responses. We focus on two contextual enhancement methods: few-shot learning (FSL) and Retrieval-Augmented Generation (RAG).

Few-shot learning (FSL) is a machine learning architecture in which pre-trained models are trained with minimal labeled examples to achieve high-accuracy predictions. FSL is especially suitable for classification tasks where collecting adequate training datasets is difficult. In contrast, *Retrieval-Augmented Generation* (RAG) enhances the output of LLM by leveraging authoritative external knowledge bases. This method extends LLM capabilities by enabling access to specific domains or internal knowledge bases without necessitating model retraining.

Table 6.2: Comparison between FSL and RAG

No.	Input	FSL Output	RAG Output
1	What types of EXIF metadata does the code block below refer to? `public class ExifMetadataManager { public static void main(String[] args) { final e[] exifMetadataArray = { new e("Make", 271, 2), new e("Software", 305, 2), new e("DateTime", 306, 2), new e("GPSInfo", 34853, 4) }; Image image = new Image("input.jpg"); addExifMetadata(image, exifMetadataArray); } ... }`	**Make (tag 272, type 2)**: This metadata tag stores the manufacturer or brand of the camera or device that captured the image. **Software (tag 305, type 2)**: This tag indicates the name and version of the software or firmware of the camera or input device that was used to process the image. It is also a string type. **DateTime (tag 306, type 2)**: This tag records the date and time when the image was created or last modified. The format is "YYYY:MM:HH:MM". **GPSInfo (tag 34853, type 4)**: This tag contains a set of sub-tags that provide detailed GPS information about the location where the image was captured. It includes various types of information such as latitude, longitude, altitude, and other GPS-related data. This tag is of type IFD (Image File Directory), which can contain multiple sub-tags.	**Make (tag 272, type 2)**: This metadata tag stores the manufacturer or brand of the camera or device that captured the image. **Software (tag 305, type 2)**: This tag indicates the name and version of the software or firmware of the camera or input device that was used to process the image. It is also a string type. **DateTime (tag 306, type 2)**: This tag records the date and time when the image was created or last modified. The format is "YYYY:MM:HH:MM". **GPSInfo (tag 34853, type 4)**: This tag contains a set of sub-tags that provide detailed GPS information about the location where the image was captured. It includes various types of information such as latitude, longitude, altitude, and other GPS-related data. This tag is of type IFD (Image File Directory), which can contain multiple sub-tags.
2	What types of EXIF metadata does the code block below refer to? `public class ExifMetadataManager { public static void main(String[] args) { final e[] exifMetadataArray = { new e("Maker", 271, 2), new e("Software", 305, 2), new e("DateTime", 306, 2), new e("GPSInfoIFDPointer", 34853, 4) }; Image image = new Image(" input.jpg"); addExifMetadata(image, exifMetadataArray); } ... }`	**Maker (tag 271, type 2)**: *I'm not entirely sure what 'Maker' refers to—it could be related to the camera manufacturer such as Nikon, Canon, Sony, etc. or another EXIF metadata term.* **Software (tag 305, type 2)**: This tag provides information about the software used to process or edit the image. **DateTime (tag 306, type 2)**: Represents the date and time when the image was created or modified. **GPSInfoIFDPointer (tag 34853, type 4)**: *It seems to relate to GPS location data in the image, but I'm uncertain about its exact meaning.*	**Maker (tag 271, type 2)**: This tag is often used to store the name of the manufacturer of the camera or input device used to capture the image. It is a string type. **Software (tag 305, type 2)**: This tag indicates the name and version of the software or firmware of the camera or input device that was used to process the image. It is a string type. **DateTime (tag 306, type 2)**: This tag records the date and time when the image was created or last modified. The format is "YYYY:MM:DD HH:MM:SS". **GPSInfoIFDPointer (tag 34853, type 4)**: This tag is a pointer to the GPS information IFD (Image File Directory), which contains detailed GPS information about the location where the image was captured. It is of type IFD (Image File Directory), which can contain multiple sub-tags that store various types of GPS-related data such as latitude, longitude, altitude, and other geographic information.

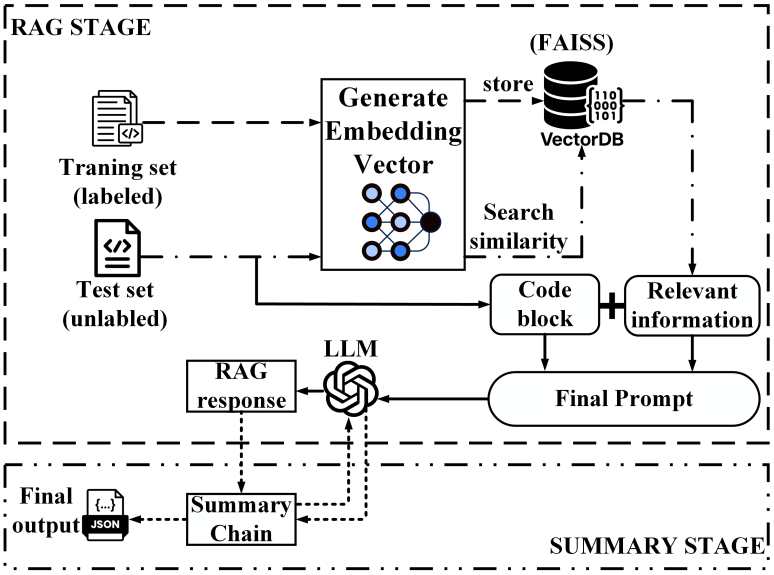

Figure 6.1: Code summarization for apps in the Self-developed group

Although some studies [35], [66] have already demonstrated that RAG outperforms FSL in tasks requiring extensive knowledge without requiring many parameters, we decide to evaluate both approaches to ensure that RAG is the right choice also for tasks requiring the evaluation of source code designed to handle EXIF metadata. In particular, we consider two examples, as represented in Table 6.2, where we ask the LLM (GPT-4)[9] to explain which types of EXIF metadata the input code blocks are related to. We prompt the LLM with two different inputs corresponding to rows 1 and 2 of Table 6.2. The main difference between the two inputs is that row 1 uses *"Make"* and *"GPSInfo"*, while row 2 uses *"Maker"* and *"GPSInfoIFDPointer"*. In practice, these parameters are not significantly different, and both are used to store information about the camera manufacturer and GPS location. However, the outputs of FSL and RAG show significant differences. Specifically, for row 2, using *"Maker"* and *"GPSInfoIFDPointer"* outside the learning examples confused the FSL, leading to inconsistent responses and an inability to provide a definitive answer. In contrast, leveraging external knowledge, RAG produces more consistent outputs across the two examples. Therefore, we conclude that RAG is more suitable for the ALIBIS use case.

6.2.2.2 Retrieval-Augmented LLM

Figure 6.1 depicts the code summarization flow for a target code block A_i belonging to an app in the Self-Developed Group, that is, A_i exploits self-developed functions. The process consists of two phases: the RAG stage and the Summary stage, which are described in the following sections.

[9] https://platform.openai.com/docs/models

a) RAG stage

In general, RAG operates according to four main phases: (1) *Creating External Data*: converting diverse data sources into numerical representations using embedding models and storing them in a vector database; (2) *Retrieving Relevant Information*: transforming user queries into vectors and matching them with the vector database to extract relevant information; (3) *Augmenting the LLM Prompt*: employing prompt engineering to integrate user input with retrieved data, forming an augmented prompt for the LLM; and (4) *Updating External Data* (optional step): regularly updating external data through automated or batch processes to ensure current information for retrieval [88].

For the first step, i.e., *creating the external dataset*, we have to select an external data source that is relevant to our domain, then transform them into numerical representations utilizing embedding models, and finally store them in a vector database. As a data source, we consider the subset of the apps in the LAD dataset whose code blocks exploit self-developed functions, denoted as LAD_{self_kb} (see Section 6.4.1 for more details).

More precisely, we consider the set of code blocks of the apps in LAD_{self_kb}, to which we refer as CB_{self_kb}. Then, we encode each code block $B_j \in CB_{self\_kb}$ in an embedding vector through the embedding model f_e:

$$v_{B_j} = f_e(B_j), \quad \forall B_j \in CB_{self\_kb}$$

As f_e we select the default embedding model provided by the adopted LLM. All embedding vectors of blocks in CB_{self_kb} are then stored into a vector database, denoted as V_{RAG}.

An example of this step is illustrated in Table 6.3, where row 1 represents an example of code blocks, whereas row 2 shows the corresponding embedding vector.

The second phase implies *retrieving relevant information* to augment the query to be submitted to LLM, aka LLM prompt. For this purpose, we have to transform the original LLM prompt into vectors and match them with the vector database to extract relevant information. In our scenario, the prompt/query is the target app $\mathcal{A}$ to be labeled. In particular, we want to exploit LLM to associate a label with each of $\mathcal{A}$'s code blocks that contain self-developed functions. Thus, as a prompt, we give each of them to LLM separately. In the following, we describe the process for a single code block $A_i \in \mathcal{A}$. We transform A_i into an embedding vector (i.e., V_{A_i}) through the same embedding model f_e provided by the target LLM, that is $v_{A_i} = f_e(A_i)$.

The retrieval process matches v_{A_i} with vectors in V_{RAG} and selects the vector $\hat{v} \in V_{RAG}$ that is the most similar to v_{A_i} based on a similarity measure (e.g., cosine similarity). Vector $\hat{v}$ represents the relevant information to augment the LLM prompt.

An example of this process is presented in Table 3, where column 2 contains a code block A_i (row 1) and the corresponding embedding vector (row 2), whereas row 3 displays the relevant information retrieved from the vector database V_{RAG}.

In the third phase, we *augment the LLM prompt* with the retrieved vector $\hat{v}$. Therefore, the augmented prompt p_{aug} is generated as:

$$p_{\text{aug}} = A_i + \hat{v}$$

The augmented prompt p_{aug} is then used to request the LLM to perform code summarization. The response returned by the LLM is referred to as the RAG response.

JSON 6.1: Self-Developed Group Output

```
{
    "app_name": "App A",
    "code_block_index": "i"
    "metadata": {
        "datetime": "shared",
        "GPS": "shared",
        "smartphone model": "removed",
        "smartphone brand": "removed",
        "serial number": "removed",
    }
}
```

b) Summary stage

In general, LLM could return a lengthy response, where often parts of the original prompt are included in it (see, as an example, the results in Table 6.2). To better use the obtained RAG response, we implement an additional step called **Summary Stage**. In particular, we exploit the Langchain agent[10] to summarize the RAG response. Thanks to the Langchain agent, we can create output templates for LLM, for example, specifying the desired keys in the JSON output and removing unnecessary information. We define JSON output templates so to return labels in the format $\langle$*metadata type*, *shared*/*removed*$\rangle$ (cf. JSON 6.1).

[10] https://python.langchain.com/v0.2/docs/tutorials/summarization/

Table 6.3: RAG Example

<table>
<tr><th>No.</th><th>Code Block $B_j \in CB_{self\_kb}$</th><th>Code Block A_i</th></tr>
<tr><td>1</td><td>

```
public class ExifMetadataManager {
    public static void main(String[] args) {
        final e[] exifMetadataArray = {
            new e("Make", 271, 2),
            new e("Model", 272, 2),
            new e("Software", 305, 2),
            new e("DateTime", 306, 2),
            new e("GPSInfoIFDPointer",
            34853, 4),
        };
        Image image = new Image("input.jpg");
        addExifMetadata(image,
        exifMetadataArray);
        image.displayInfo();
    }
    public static void addExifMetadata
    (Image image, e[] exifMetadataArray)
    {
        for (e metadata : exifMetadataArray)
        {
                image.addMetadata(
                metadata.getName(),
                metadata
            );
        }
    }
}
```

</td><td>

```
public class ImageMetadataHandler {
    public static void main(String[] args) {
        final Metadata[] cameraSpecsArray =
        {
            new Metadata("ImageHeight",
            258, 3, 4),
            new Metadata("ImageWidth",
            259, 3, 4),
            new Metadata("Maker", 271, 2),
            new Metadata("Model", 272, 2),
            new Metadata("SoftwareVersion",
            306, 2),
            new Metadata("CaptureDate", 307, 2),
        };
        final Metadata[] gpsMetadataArray = {
            new Metadata("Latitude", 34855, 5),
            new Metadata("Longitude", 34856, 5)
        };
        Image image = new Image("output_image.jpg");
        addImageMetadata(image, cameraSpecsArray);
        addImageMetadata(image, gpsMetadataArray);
        image.displayMetadata();
    }

    public static void addImageMetadata(Image image,
    Metadata[] metadataArray) {
        for (Metadata metadata : metadataArray) {
            image.addMetadata(metadata.getName(),
            metadata);
        }
    }
}
```

</td></tr>
<tr><td></td><td>Embedding Vector for B_j</td><td>Embedding Vector for A_i</td></tr>
<tr><td>2</td><td>

```
[0.010854944001375403, 0.015865415840328054,
...
-0.010467036960814668, 1.551882500625406e-05]
```

</td><td>

```
[0.00906585767962255, 0.02164066707644313,
...
-0.018437951003234927, 0.0004406128099815989]
```

</td></tr>
<tr><td></td><td colspan="2">Relevant Information extracted from V_{RAG} for A_i</td></tr>
<tr><td>3</td><td colspan="2">

```
public class ExifMetadataManager {
    public static void main(String[] args) {
        final e[] exifMetadataArray = {
            new e("Make", 271, 2),
            new e("Model", 272, 2),
            new e("Software", 305, 2),
            new e("DateTime", 306, 2),
            new e("GPSInfoIFDPointer",
            34853, 4),
        };
        Image image = new Image("input.jpg");
        addExifMetadata(image,
        exifMetadataArray);
        image.displayInfo();
    }
    public static void addExifMetadata
    (Image image, e[] exifMetadataArray)
    {
        for (e metadata : exifMetadataArray)
        {
                image.addMetadata(
                metadata.getName(),
                metadata
            );
        }
    }
}
```

</td></tr>
</table>

6.3 ALIBIS Architecture & Implementation

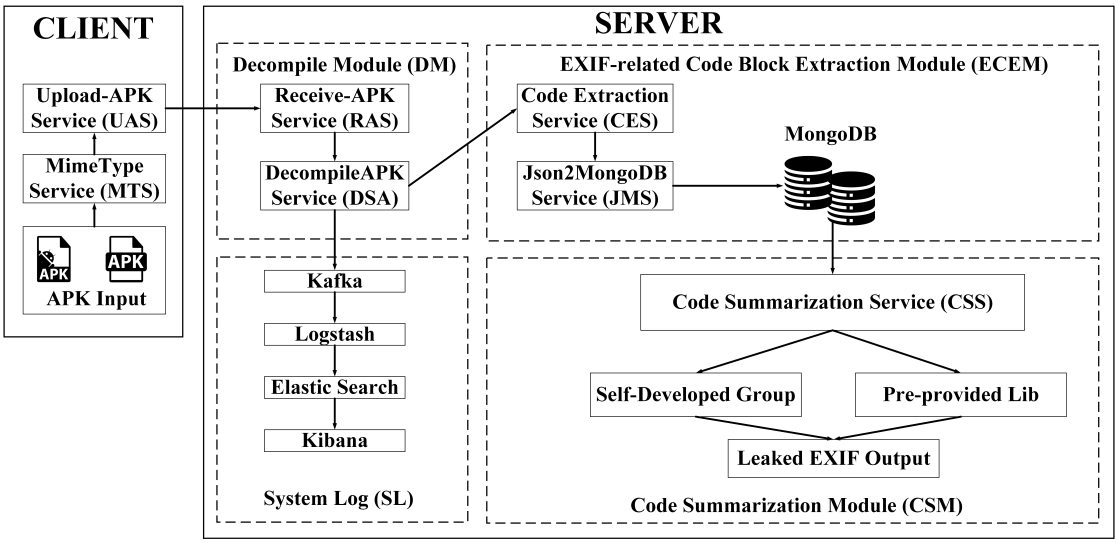

Figure 6.2: ALIBIS Architecture

ALIBIS has been designed to receive an app's APK as input, through a client, and return the app's labels associated with the app's code blocks. It comprises five main modules (see Figure 6.2), all programmed in Python. In the following section, we present the ALIBIS client and server modules.

6.3.1 Client

The client module aims to upload the app to the ALIBIS server for analysis. This is done only if the app indeed has the capabilities to open (i.e., access images in public storage) and upload/share images. Therefore, the client evaluates whether the target app requests the *metadata permissions*. It is relevant to note that having these permissions does not necessarily mean that the app is able to access and share images. Indeed, the permissions are restricted to specific file types, such as only PDFs, as specified by the mimeType[11] parameter in the Manifest.xml. As such, given the target app, the MimeType Service (MTS) extracts the permissions from the Manifest.xml and checks if their mimeType is *"image/*"* and/or *"*/*"*.

If the app has these capabilities, its APK file has to be uploaded to the ALIBIS server. For this, we implement the Upload-APK Service (UAS) on the client-side and Receive-APK Service (RAS) on the ALIBIS server using gRPC[12] instead of traditional HTTP/FTP protocols. The gRPC, an open-source RPC developed by Google, facilitates client-server communication in distributed systems, using HTTP/2 for transport and

[11] https://developer.android.com/reference/androidx/media3/common/MimeTypes
[12] https://grpc.io/

Protocol Buffers (protobuf) for interface description. Previous studies [51] show that gRPC operates at twice the speed of HTTP.

6.3.2 Decompile Module (DM)

The goal of the Decompile Module (DM) is to extract the app's source code from its corresponding APK file. To obtain the app's source code, we utilize Apktool[13] and Dex2Jar[14] to decompile APK files. Although these tools have been employed in many previous studies (e.g., [70]), they are command-line tools requiring setting up a testing environment (Windows or Linux) and manually entering commands for each app. To eliminate this inconvenience, we upgrade Apktool and Dex2Jar and package these tools into a Docker container, i.e., the Decompile APK service (DSA). This enables the automatic decompiling of APK files. The DM's output includes Java and interface-related files. Additionally, we log the start and end times of the APK decompilation process into the System Log (SL) to facilitate the evaluation of ALIBIS performance (cf. Section 6.4.4).[15]

6.3.3 EXIF-related Code Block Extraction Module (ECEM)

The goal of this module is to extract all EXIF-related code blocks from the app's source code.

It plays a key role in ALIBIS, as it lays the foundation for accurate code summarization in the next module. In general, ECEM goes through two main steps. In the first step, ECEM extracts all code files containing functions/methods used to process EXIF metadata, done by the Code Extraction Service (CES). Then, in step 2, ECEM extracts EXIF-related code blocks from the code files obtained from step 1.

Given a target app, in step 1, CES scans its source code, excluding the interface-related files, and removes developer comments. It then selects among the app's source code only those code files containing the keyword *"exif"*. Additionally, if the selected code files contain keywords *"android.media.ExifInterface"*, *"androidx.exifinterface.media"*, CES classifies the target app as belonging to the Pre-provided Lib Group. Otherwise, the app belongs to the Self-Developed Group.

Then, in step 2, to extract the EXIF-related code blocks, CES parses the extracted code files (obtained in step 1), looking for a set of specific keywords. While the process is similar for apps in the Pre-provided Lib Group and Self-Developed Group, CES uses two different lists of keywords for code extraction.

In particular, since apps in the Pre-provided Lib Group use Google's built-in classes, the list of keywords, denoted as Lib_{Keys}, includes the Java methods for handling EXIF metadata declared by Google (e.g., *getAttribute()* for retrieving all values from the image EXIF metadata, *getDateTime()* for extracting the datetime information); class-

[13] https://apktool.org/

[14] https://github.com/pxb1988/dex2jar

[15] We exploit Kafka (https://kafka.apache.org/), Elasticsearch, Logstash, and Kibana (https://www.elastic.co/elastic-stack) to store and display logs.

specific constants, such as *TAG_GPS_LONGITUDE, TAG_MAKE, TAG_MODEL*, etc. Moreover, since EXIF metadata values are often stored in arrays during programming, all terms related to Java array declarations, such as *ArrayList, LinkedList, HashMap, HashSet, and StringBuilder*, are also included in Lib_{Keys}.

In contrast, apps in the Self-Developed Group use self-developed functions; thus, we cannot rely on pre-determined method names/class-specific constants. However, we can identify all Java array handling methods. As such, we prepare a different keyword list, denoted as $Self_{Keys}$, including terms relatd to Java array declarations and handling methods (e.g., ArrayList‘, ‘LinkedList‘, ‘HashMap‘, ‘HashSet‘, and ‘StringBuilder, *‘add()‘, ‘get()‘, ‘set()‘, ‘remove()‘, ‘clear()‘, ‘put()‘, ‘access()‘, ‘contains()‘, ‘append()‘, ‘delete()‘, and ‘insert()‘*) Additionally, $Self_{Keys}$ also includes keywords related to sensitive metadata, such as *‘DateTime‘, ‘GPS‘, ‘Latitude‘, ‘Longitude‘, ‘Altitude‘, ‘Model‘, ‘Make‘, ‘Maker‘, and ‘Software‘*.

By using Lib_{Keys} and $Self_{Keys}$, CES scans the source code of the selected apps. If a keyword is found, the code is parsed to identify the function by determining the nearest enclosing curly braces and extracting the corresponding code block. The extracted EXIF-related code blocks are stored in a structured JSON format and returned as output.

Finally, the Json2MongoDB Service (JMS) (see Figure 6.2) imports the resulting EXIF-related code blocks from the JSON file into the MongoDB database.

Thanks to the two-step EXIF-related code blocks extraction mechanism described above, ALIBIS mitigates the limitations of static analysis caused by code obfuscation. Specifically, in step 1, code files extraction, the keywords such as *“exif"*, *“android.media.ExifInterface"* and *“androidx.exifinterface.media"* correspond to import commands to use the EXIF library and are not affected by code obfuscation. In addition, while code obfuscation can change the function name after decompilation, the Lib_{Keys} and $Self_{Keys}$ used in step 2, including constants, Java array declarations, and native handling methods, are also not impacted by obfuscation, as shown in the Example-code 6.2. Specifically, although the function name has changed to E() (i.e., lines 23-38 of Example-code 6.2) and its purpose cannot be determined because of the obfuscated name, we still understand that this is the function used to add GPS information to an array through the array declaration (*array3*, lines 4-20 of Example-code 6.2) and the *gpsTag.set()* (line 31 of Example-code 6.2) method used in E().

6.3.4 Code Summarization Module (CSM)

The Code Summarization Service (CSS) performs code summarization using the approaches described in Section 6.2. For the Pre-provided Lib Group, we use the Selenium framework[16] to automatically crawl EXIF method names and corresponding descriptions from Google documentation, needed to construct ⟨method_name, label⟩ pairs (see Section 6.2.1). Next, we use the *code2vec* library[17] to create the ASTs, then convert the

[16]Selenium is an open-source automation tool widely used for web browser interaction. Selenium is instrumental in web scraping tasks because it can simulate fundamental user interactions across multiple browsers.https://www.selenium.dev/

[17]https://code2vec.org/

Table 6.4: Comparison between OpenAI API and Code Llama

Task	OpenAI API	Code Llama
RAG stage	29.7 s	78s
Summary stage	1.93 s	11.7 s

EXIF-related code blocks into embedding vectors and compute cosine similarity.

For the Self-Developed Group, we develop the RAG Stage and Summary Stage using the Langchain Framework. LangChain is an open-source framework designed to facilitate the development of apps utilizing LLM. It supports creating complex applications such as advanced prompt engineering, chatbots, data analysis tools, etc.

LangChain plays a role as an API endpoint, enabling connections to various LLM. Switching LLM can be done easily by changing the API key in the ALIBIS source code. For the ALIBIS implementation, we have considered two main models: OpenAI API[18] and Code Llama.[19] The OpenAI API is a cloud-based service that allows developers to access OpenAI's state-of-the-art language models, such as GPT-3.5 and GPT-4. The API service provided by OpenAI is a paid service based on input (query prompt) and output (LLM's response) tokens. Conversely, Code Llama, developed by Meta, is a free specialized Llama model version designed for code-related tasks like code generation, completion, and bug detection. Adopting an open-source LLM to be deployed locally (like Code Llama) could mitigate concerns about sharing the app's source code with third parties, and reduce the cost of paid service (like OpenAI). However, due to hardware limitations in our lab,[20] we decide to test both Code Llama and OpenAI to select the model that best meets our requirements. In particular, we deploy Code Llama on a local server with the following configuration: an Intel Core i9-12900H CPU (20 cores), 32GB RAM, and a GeForce 3070Ti GPU with 16GB of memory. For OpenAI, we utilize their API to access the cloud-based service.

We conduct manual tests with 10 EXIF-related code blocks as the input to compare the performance of OpenAI API and Code Llama. Regarding accuracy, the OpenAI API provides more reliable and consistent responses than Code Llama. Regarding the response speed, the OpenAI API significantly outperforms Code Llama. Table 6.4 details the average response times of the OpenAI API and Code Llama for 10 input code blocks. For both stages—RAG and summary—the OpenAI API consistently responds faster than Code Llama, with a total execution time approximately eight times quicker. Therefore, we use the OpenAI API (model *GPT-4*) for ALIBIS implementation and evaluation. The total cost for summarizing the code for the *LAD* dataset (2190 apps) was $200 USD.

[18] https://openai.com/index/openai-api/

[19] https://ollama.com/library/codellama

[20] Code Llama requires very demanding hardware configuration; for example, deploying a 405 billion-parameter model requires at least 256GB of RAM, 1944GB of GPU, and 780GB of storage.https://llamaimodel.com/requirements/.

Example-code 6.2: code obfuscation

```
import androidx.exifinterface.media.ExifInterface;
private static final d[] array3;
static {
    array3 = new d[] {
        new d("GPSVersionID", 0, 1), new d("GPSLatitudeRef", 1, 2),
        new d("GPSLatitude", 2, 5, 10), new d("GPSLongitudeRef", 3, 2),
        new d("GPSLongitude", 4, 5, 10), new d("GPSAltitudeRef", 5, 1),
        new d("GPSAltitude", 6, 5), new d("GPSTimeStamp", 7, 5),
        new d("GPSSatellites", 8, 2), new d("GPSStatus", 9, 2),
        new d("GPSMeasureMode", 10, 2), new d("GPSDOP", 11, 5),
        new d("GPSSpeedRef", 12, 2), new d("GPSSpeed", 13, 5),
        new d("GPSTrackRef", 14, 2), new d("GPSTrack", 15, 5),
        new d("GPSImgDirectionRef", 16, 2), new d("GPSImgDirection", 17, 5),
        new d("GPSMapDatum", 18, 2), new d("GPSDestLatitudeRef", 19, 2),
        new d("GPSDestLatitude", 20, 5), new d("GPSDestLongitudeRef", 21, 2),
        new d("GPSDestLongitude", 22, 5), new d("GPSDestBearingRef", 23, 2),
        new d("GPSDestBearing", 24, 5), new d("GPSDestDistanceRef", 25, 2),
        new d("GPSDestDistance", 26, 5), new d("GPSProcessingMethod", 27, 7),
        new d("GPSAreaInformation", 28, 7), new d("GPSDateStamp", 29, 2),
        new d("GPSDifferential", 30, 3), new d("GPSHPositioningError", 31, 5)
    };
}
private void E() {
    for (int i = 0; i < this.f.length; ++i) {
        for (final Map.Entry<String, c> entry : this.f[i].entrySet()) {
            final String tagName = entry.getKey();
            final c cTag = entry.getValue();
            if ("GPSLatitude".equals(tagName) || "GPSLongitude".equals(tagName) || "GPSAltitude".equals(tagName)) {
                for (d gpsTag : array3) {
                    if (gpsTag.tagName.equals(tagName)) {
                        gpsTag.set(cTag.j(this.h));
                        Log.d("ExifInterface", "Set " + gpsTag.tagName + " = " + gpsTag.get());
                    }
                }
            }
        }
    }
}
```

6.4 Experimental Evaluation and Mitigation strategies

We conduct several tests to evaluate ALIBIS's performance. In particular, to evaluate its effectiveness, we measure its accuracy in two distinct experiments. The first aims to assess separately the two proposed code summarization strategies (cf. Section 6.2). As such, we measure the accuracy that ALIBIS reaches for each separate group of apps (Pre-provided Lib Group and Self-Developed Group). The second experiment aims to evaluate the overall system performance over the whole dataset using K-fold cross-validation. Moreover, we evaluate its efficiency by assessing ALIBIS's execution time for all core modules. In this section, we also present a possible mitigation strategy to avoid EXIF metadata leakage. Before presenting the experiments, in the next section, we introduce the dataset used to evaluate ALIBIS.

6.4.1 Dataset

As introduced in Section 6.2, we exploit the MetaLeak's result (cf. Chapter 5) as a knowledge base for the code summarization approaches (i.e., for computing code similarity and as an external source for RAG). MetaLeak labeled dataset consists of 5,000 apps downloaded in March 2023 from the app store APKcombo,[21] selected based on their number of installations. The categories of downloaded apps are distributed as follows: Photography (79.74%), Beauty & Art (11.4%), Tools (4.52%), Productivity (2.06%), Communication (1.12%), Entertainment (0.36%), Business (0.32%), Personalization (0.30%), Lifestyle (0.10%), and Social (0.08%). Among 5,000 apps, 21.9% ($\approx$ 1095) were found to leak at least one of the five types of considered sensitive metadata. Specifically, MetaLeak employs hybrid analysis to monitor the app's sent-out traffic, then categorizes the app as either leaking or non-leaking. Additionally, MetaLeak returns the type of leaked metadata (i.e., the five types of sensitive metadata that we are focusing on) in the case of an app identified as leaking. However, MetaLeak's results only label at the app level instead of annotating the code block level. Thus, we use the ECEM module to extract EXIF-related code blocks from the app's source code (cf. Section 6.3.3). We perform the code extraction step for the entire dataset. Finally, we manually annotate the obtained EXIF-related code blocks with the assistance of ten graduate students experienced in Android programming.

It is important to emphasize that MetaLeak's labels have a format slightly different from the labels introduced in Section 6.2, that is, ⟨metadata type, shared/removed⟩. For example, in MetaLeak, *leaking apps* that expose GPS and datetime information have an actual label of ["Leak GPS", "Leak datetime"], whereas *non-leaking apps* have an actual label of ["No leak"]. These labels have been easily transformed into the

[21] APKcombo is an app store that supports downloading APK files using the package name (a unique identifier for the app) instead of the app name, which can reduce confusion with apps having similar names. This app store can be accessed from anywhere in the world and supports MetaLeak's auto-download module, which is based on Selenium. https://apkcombo.com/

⟨metadata type, shared/removed⟩ format.[22]

Moreover, to avoid classification bias due to an imbalance between the number of leaking (1095) and non-leaking (3905) apps, starting from the MetaLeak dataset, we generate a Labeled Apps Dataset, LAD, consisting of the 1095 apps, detected to leak metadata, plus another 1095 apps that do not leak metadata. These latter have been selected from the 3905 non-leaking apps based on popularity (i.e., the number of installations). Thus, LAD dataset comprises 2190 apps, where 1346 apps, LAD_{lib}, belongs to the Pre-provided Lib Group,[23] whereas 844 apps, LAD_{self}, belongs to Self-Developed Group.

Based on the type of experiment, we split the LAD dataset into different portions. In particular, to test the code summarization strategy, we divide the LAD_{lib} and LAD_{self} into two parts: 50% for the knowledge base, LAD_{lib_kb} and LAD_{self_kb}, and 50% for testing. In contrast, for the overall evaluation with K-fold cross-validation, the LAD dataset is divided into 5 folds. With 1095 testing apps (~ 55GB) as input, we obtain (~ 110GB) of source code after performing decompilation with the Decompile Module (cf. Section 6.3.2). Subsequently, we extract EXIF-related code blocks from the 110GB of source code and obtain 175MB of source code about EXIF metadata (cf. Section 6.3.3). Finally, EXIF-related code blocks related to apps in the testing dataset, i.e., LAD_{te}, are processed by the Code Summarization Module (CSM). The results obtained are used for evaluations, as described in the next sections.

6.4.2 Code summarization performance

To test the accuracy of code summarization strategies, we need to compare the labels in the MetaLeak dataset, hereafter the *actual labels*, with the labels returned by ALIBIS code summarization, aka *predicted labels*. As introduced in Section 6.2, both code summarization strategies return five types of labels in the format ⟨metadata type, shared/removed⟩, indicating for each one of the considered five metadata types, whether the app is sharing/removing it. Based on returned labels, we could have two cases: $\mathcal{A}$ does not leak any sensitive metadata, that is, all five labels indicate that metadata are removed, we denoted this event as l_0; or $\mathcal{A}$ leaks some or all of the five metadata type, we denote this event as $l_+ \subseteq \{l_1, l_2, l_3, l_4, l_5\}$, where l_j indicate the leaked metadata type. Thus, given an app $\mathcal{A}$ we can formalize its predicted label as $Predict(\mathcal{A}) \in \{pl_o, pl_+\}$, where $pl_+ \subseteq \{l_1, l_2, l_3, l_4, l_5\}$. Similarly, its actual label can be represented as $Actual(\mathcal{A}) \in \{al_o, al_+\}$, where $al_+ \subseteq \{l_1, l_2, l_3, l_4, l_5\}$.

To estimate the accuracy, we consider the following metrics (see Table 6.5): *True Positives (TP):* the target app is labeled by the code summarization as leaking some or all types of sensitive metadata, and it indeed leaks one or more of them; *False Positives (FP):* the target app is labeled as leaking some or all sensitive metadata, but it does not actually leak any sensitive metadata; *True Negatives (TN):* the target app is labeled

[22] We transform the MetaLeak labels to align with the JSON format produced by the summary chain (cf. JSON 6.1), thereby facilitating direct comparison.

[23] In particular, 984 apps exploit method-1 -*android.media.ExifInterface* and 362 method-2 -*androidx.exifinterface.media*

Table 6.5: True Positive, False Positive, True Negative, False Negative

TP	$Actual(\mathcal{A}) = al_+ \wedge Predict(\mathcal{A}) = pl_+ \wedge al_+ = pl_+$
FP	$Actual(\mathcal{A}) = al_0 \wedge Predict(\mathcal{A}) = pl_+$
TN	$Actual(\mathcal{A}) = al_0 \wedge Predict(\mathcal{A}) = pl_0$
FN	$Actual(\mathcal{A}) = al_+ \wedge Predict(\mathcal{A}) = pl_0$

as not leaking sensitive metadata, and it indeed does not leak any sensitive metadata; *False Negatives (FN):* the target app is labeled as not leaking sensitive metadata, but it actually leaks some or all sensitive metadata.

Specifically, for apps in the Pre-provided Lib Group the ALIBIS code summarization strategy obtains a confusion matrix: TP = 357, FP = 18, FN = 51, TN = 247. For those apps in Self-Developed Group: TP = 252, FP = 28, FN = 35, TN = 107. The ALIBIS code summarization strategy performs well in both groups, achieving a high number of True Positives (TP) and True Negatives (TN). Although, compared to the Self-Developed Group, the Pre-provided Lib Group shows a slightly higher False Negative (FN) rate but a lower False Positive (FP) rate, indicating that ALIBIS maintains a balance in both code summarization strategies.

Based on the confusion matrix metrics, we also calculate *Precision, Recall*, *F1 Score*, and *Accuracy* for the two groups, as reported in Table 6.6. These results indicate that the Pre-provided Lib Group's code summarization strategy (i.e., mapping function and code similarity) slightly outperforms the one for the Self-Developed Group. Specifically, the Pre-provided Lib Group's strategy offers fewer false positives and can identify true positives better. These results can be motivated by using Google documentation, which, as expected, is more accurate than LLM. However, we emphasize that we can rely on Google documents only for a portion of apps in the Pre-provided Lib Group. Therefore, the code summarization method using LLM is more generalizable, especially when documentation is unavailable. Finally, we can conclude that LLM have great potential in estimating the risk of sensitive metadata leakage.

Table 6.6: True Positive, False Positive, True Negative, False Negative

	Pre-provided Lib Group	**Self-Developed Group**
Accuracy	0.897	0.851
Precision	0.952	0.900
Recall	0.875	0.878
F1 Score	0.912	0.889

False positive, False negative cases. We further analyze the cases of false positive and false negative. Indeed, since ALIBIS relies on code summarization, we want to verify if it could be affected by the presence of deprecated functions/code, a common weakness in static analysis solutions. This could happen if the target app's source code contains EXIF-related code blocks that are not actually executed at run-time. For simplicity,

assuming an app with just a code block removing EXIF metadata that is not executed at run time (Actual(A)=al_+), ALIBIS could wrongly label as Predict(A)=pl_0, leading to a false negative event. Similarly, it could be the case of a code block sharing EXIF metadata that is not executed at runtime (Actual(A)=al_o), ALIBIS could wrongly label it as Predict(A)=p_+, leading to a false positive event. We first manually analyze the FP cases. These are all due to the confusion of the datetime taken (i.e., the time the photo was taken) with the date modified. Specifically, 46 apps did not share in their sent-out traffic the datetime taken but instead shared the date modified. This happens when the user has made edits to the original photo after taking it (e.g., increasing the contrast) and before sharing it. ALIBIS labels these as leaking apps. However, even if the date modified is not the sensitive metadata initially targeted in this study, it still represents sensitive data; thus, having ALIBIS be able to detect it (even as a false positive) is a positive aspect. Additionally, it is worth noting that these false positives are not due to deprecated code.

We then analyzed the 86 apps that fell into the FN case. The main cause, affecting 76 apps, is related to EXIF-associated code blocks for GPS that contain an excessive number of parameters (e.g., *array3*, lines 4–20 of Example-code 6.2). Among these parameters, only three—*GPSLatitude, GPSLongitude*, and *GPSAltitude*—actually represent GPS coordinates. Because the remaining parameters are not true coordinates, the LLM often becomes distracted by them, which leads to hallucinations and incorrect code summarization. In particular, when the three real GPS-coordinate fields contain meaningful values in the sent-out traffic (indicating an actual GPS leakage), the LLM sometimes focuses instead on unrelated GPS-like fields (e.g., *GPSDestDistance*) and misinterprets them as irrelevant or non-sensitive. As a result, the model fails to detect that the app truly leaks GPS metadata. After aggregating all code block-level predictions, the app is ultimately labeled as not leaking GPS, yielding an app-level false negative (FN). An excessive number of GPS parameters is the most common cause of FN, with 47 apps in the Pre-provided Lib Group and 29 in the Self-Developed Group. Additionally, among 86 FN cases, only 10 apps have been wrongly labeled due to deprecated code, with 4 apps in the Pre-provided Lib Group and 6 apps in the Self-Developed Group. In particular, although these apps contain code blocks that remove sensitive metadata, in reality (when tested with hybrid analysis), this function is not executed at run-time.

To overcome these limits, as future work, we plan to integrate FlowDroid [9] in ALIBIS. FlowDroid is a static taint analysis tool for Android that examines data flows from sources (e.g., the flow starting points) to sinks (e.g., network or file operations) by modeling the Android lifecycle and callback methods. Specifically, after extracting the EXIF-related code blocks, we will refer to them as *source* (i.e., the starting point of sensitive metadata). Then, we will identify the *sinks*, i.e., actions that transmit data to the Internet (e.g., linking to HTTP clients). Using this method, we will exclude deprecated functions if no link is found between the source and sink. Moreover, to avoid possible hallucination cases related to GPS parameters, as future work, we will create a list of examples and embed them directly into the input prompt to guide LLM. Specifically, we will create specific definitions for each GPS parameter, for example,

{GPSVersionID: the version of GPS information (e.g., “2.2.0.0"); GPSLatitude: latitude value, typically as three numbers: degrees, minutes, seconds; GPSLatitudeRef: Latitude direction: “N" (North) or “S" (South); etc.}, to enhance context for LLM.

6.4.3 K-fold Cross Validation

To evaluate the overall performance of ALIBIS, we use k-fold cross validation with $k = 5$. Therefore, we split the ALIBIS dataset (2,190 apps) into 5 folds (fold-1, fold-2, fold-3, fold-4, fold-5), each containing 438 apps. First, we use fold-1 as the testing set and the remaining folds (fold-2, fold-3, fold-4, fold-5) as the training set (aka knowledge base). Next, we use fold-2 as the testing set and the rest as the training set. This process is repeated 5 times. Similarly to the experiments in Section 6.4.2, we calculate accuracy, precision, recall, and F1 score for each fold.

Table 6.7: K-fold Cross-Validation Results (k=5)

Fold	**Accuracy**	**Precision**	**Recall**	**F1 Score**
Fold-1	0.869	0.891	0.885	0.888
Fold-2	0.859	0.882	0.876	0.879
Fold-3	0.872	0.898	0.885	0.891
Fold-4	0.886	0.907	0.893	0.900
Fold-5	0.857	0.873	0.866	0.869
Average	0.8686	0.8902	0.881	0.8854

Table 6.7 shows the K-fold cross-validation results. The accuracy value fluctuates between 0.857 and 0.886, demonstrating that the accuracy is stable across all folds. This means the model is reliable and performs consistently across different data sets. The precision values range from 0.873 to 0.907, demonstrating successful false positive minimization, with higher values suggesting more accurate positive predictions, while the recall values range from 0.866 to 0.893, reflecting ALIBIS's capacity to recognize true positive cases. The F1 score ranges from 0.869 to 0.900, which reflects a balance of precision and recall, highlighting the model's robustness and consistent performance across folds.

6.4.4 ALIBIS Efficiency

In the ALIBIS architecture (cf. Section 6.3), the Decompile Module (DM) is the most time-consuming step, requiring most of the system's execution time. However, the DM's execution time depends on the APK file size. By recording the start and end timestamps of the decompile process (DP) of each APK in the ALIBIS dataset, we obtain the average DP execution time relative to the APK size, as shown in Table 6.8.

In Table 6.8, the **APK size** column represents the APKs' size in megabytes (MB), the **Average decompilation time** column indicates the time required to decompile the APK file, and the **Percentage** column shows the percentage of APK files of corresponding sizes within the ALIBIS dataset. The minimum average decompilation time

Table 6.8: Decompilation time based on APK size

APK size	Percentage	Average decompilation time
size <20MB	64.17%	2.1 minutes
20MB <= size <50MB	21.82%	3.7 minutes
50MB <= size <100MB	9.06%	5.02 minutes
100MB <= size <150MB	3.06%	6.8 minutes
150MB <= size <200MB	1.06%	7.5 minutes
200MB <= size <300MB	0.59%	8.2 minutes
size >300MB	0.24%	9.12 minutes

observed is 2.1 minutes, while the maximum is 9.12 minutes. However, APKs with a size of 300MB account for a tiny proportion (0.24%) compared to those with a size of 20MB (64.17%). Leveraging the MLaaS architecture, ALIBIS can quickly scale the number of Decompile APK Services (DSA) by increasing the number of Docker containers, thereby reducing the overall decompilation time. Additionally, the average total time of the Filtering Module (FM), EXIF-related Code Bock Extraction Module (ECEM), and Code Summarization Module (CSM) is 1.5 - 2 minutes.

6.4.5 Mitigation strategies

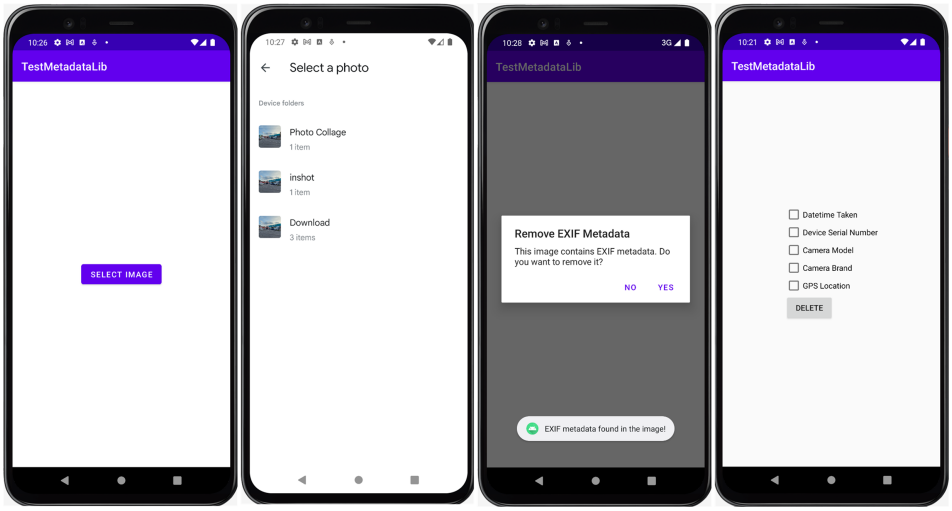

Figure 6.3: ExifMetadataLib library - user interface

As discussed in Section 3.2, current solutions for preventing sensitive metadata leakage are neither economically efficient (e.g., proxy-based solution) nor easily integrated with Android apps (e.g., ExifTool). Furthermore, our survey results (cf. Appendix A) indicate that most users do not proactively remove EXIF metadata and lack knowledge

on how to do so. Additionally, Android does not provide features to alert users of the presence of sensitive metadata or facilitate their dynamic removal from images.

We observe that regardless of the method used by an app to handle EXIF metadata, accessing images in public storage is an unavoidable step. An API query for an image is typically linked to the Open Image button on the app's user interface. This API takes the image path as input and reads the image content into a bitmap object.[24] Based on this observation, we develop ExifMetadataLib[25], with the aim of enhancing the API query image functionality. First, ExifMetadataLib checks for the presence of sensitive metadata and then issues a warning on the app's screen, as shown in Figure 6.3. It waits for user consent to retain or remove part or all of the sensitive metadata before loading the image into the bitmap object. ExifMetadataLib is lightweight and compatible with all Android versions, making its integration easy.

To evaluate the overhead implied by ExifMetadataLib integration, we tested this library in the following scenario. Since we cannot integrate ExifMetadataLib into existing apps created by other developers (e.g., Facebook app) because these apps are often equipped with anti-repackaging mechanisms [65], we choose to create our own simple app, called TestLibrary. The app allows users to open images from public storage and then upload them to cloud storage (e.g., Google Drive). More precisely, we developed a first version of the TestLibrary app without ExifMetadataLib integrated. Then, we used Android Studio Profiler[26] to measure the app's CPU and RAM during the process of users opening and uploading images. After that, we integrate the ExifMetadataLib library into the TestLibrary app and use the Android Studio Profiler again to check how much CPU and RAM the app uses when users browse images, delete all sensitive metadata, and upload the images. We repeat the test scenario as described 10 times, each time using an image with a different resolution, and then calculate the average CPU and RAM consumption. The results show that ExifMetadataLib consumes less than 1% additional CPU and less than 3MB of RAM. This small amount of overhead shows that ExifMetadataLib is a lightweight and useful tool.

[24] https://developer.android.com/develop/ui/compose/graphics/images/loading
[25] https://github.com/research-mobile-security/ExifMetadataLib
[26] https://developer.android.com/studio/profile

Chapter 7

WearLeak

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 privacy non-compliance associated with these apps. In this chapter, we focus on investigating how wearable apps manage and share sensitive data collected from end users.

As discussed in Section 2.2, wearable apps can be categorized into three types, namely, embedded, companion, and standalone. 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.[2] In general, companion apps may transmit sensitive data to their backend servers for processing or storage, and to third-party services (TPS) (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 TPS but does not specify the type of data being shared. These are instead provided in the `Data Safety` section on the Google Play console, which developers fill before app release (cf. Section 2.1). 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 chapter, we aim to investigate the *companion app's compliance with the data sharing practices declared in the Manifest and Data Safety.*

As discussed in Section 3.4, this issue has been explored in two main directions: (1) analyzing the risks arising from permission inconsistencies between companion apps and wearable devices, and (2) assessing privacy compliance by comparing collected data

[1] https://scoop.market.us/smart-wearables-statistics/
[2] https://gs.statcounter.com/os-market-share/mobile/worldwide

with developer declarations, which is the focus of this chapter. However, existing studies mainly analyze standard Android apps (i.e., non-wearable apps) and rely on traditional analysis, which suffers from scalability issues and covers only a limited set of sensitive data types (e.g., device ID, location, or network). In addition, these studies focus on only a few specific TPS (e.g., Facebook SDK) while sensitive data may be collected simultaneously by the app backend and multiple TPS. These limitations leave a significant research gap that needs to be addressed.

To address these limitations, we propose an automated method to identify all TPS integrated into a companion app and to assess privacy violations arising from the sharing of sensitive data with both TPS and the app's backend, considering 14 types of sensitive data defined in Google's documentation (cf. Table 2.1). 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 TPS 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 TPS. 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[3] can also serve as a channel for sending out sensitive data, often beyond the user's awareness [33]. 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

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

theoretical and observed behaviors. We also evaluate our approach on 711 popular companion apps and find 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 TPS, 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 TPS via deep links.

In addition, we also obtain two side results related to (1) improperly implemented authorization and (2) handling users' credentials. Specifically, as presented in Section 2.2, wearable apps offer two mechanisms for accessing health data, namely direct and indirect. However, the majority of companion apps use authorization mechanisms to collect health data indirectly. This leads to security threats if the authorization mechanism is not implemented properly. Specifically, we find that out of 287 apps using authorization mechanisms, 175 apps have access tokens with expiration times greater than 24 hours, which could allow health data to remain accessible even after the app has been uninstalled. It might be argued that this issue does not directly lead to sensitive data leaks or that the reason is the user's carelessness and lack of knowledge. However, this implementation severely violates a key provision of the GDPR, specifically *Article 5(1)(e)—storage limitation.*[4] Finally, regarding the handling of users' credentials, through observing the sent-out traffic (i.e., served behaviors), we note that 132 apps shared users' usernames and passwords in plaintext.

The remainder of this chapter is organized as follows. Section 7.1 presents our LLM-based approach for modeling theoretical behavior, whereas Section 7.2 describes prompt engineering for observed behavior. Section 7.3 presents the architecture of WearLeak Finally, Section 7.4 reports experimental results.[5]

7.1 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>`

[4]Personal data should only be stored in an identifiable form for as long as necessary for processing purposes.https://gdpr-info.eu/art-5-gdpr/

[5]The source code of WearLeak can be found on GitHub. https://github.com/research-mobile-security/WearLeak.git

with attribute com.google.android.wearable.standalone.[6] However, our analysis of 5,000 wearable apps[7] reveals that 98% do not adhere to this guideline. This also applies to TPS 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 TPS.

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 TPS 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), as described in the following.

7.1.1 GraphRAG

Retrieval-Augmented Generation (RAG) (cf. Section 2.3.3.2) 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

[6] https://developer.android.com/training/wearables/apps/standalone-apps
[7] We analyze our dataset by simply parsing 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 [30] (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).[8] 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 (cf. Section 2.3.3.3), 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.

7.1.1.1 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 2. *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>
```

[8] 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 2).

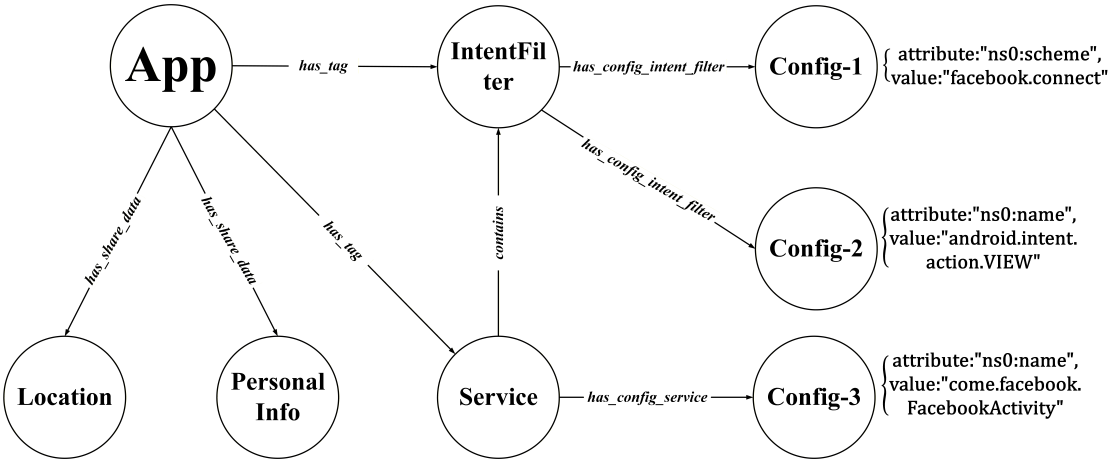

Figure 7.1: Example of $ComplianceKG_X$

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

(b) Data Safety extraction. We recall that the Data Safety lists data types that the app will collect and share. Thus, we 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 7.1 -1st row, where variables in curly braces {} are filled with X's specific values. Then, we pass the obtained prompt to *LLMGraphTransformer()*[9] method provided by Langchain to generate the KG, which is finally stored in GraphDB.

Example 3. *Let us consider the Manifest provided in Example 2 and suppose that* $X_{DataShare} = \{location, personal\ info\}$. $ComplianceKG_X$ *returned by LLMGraphTransformer() is represented in Figure 7.1.*

VectorDB - Google documentation. We use the Google documentation describing the syntax, structure, and attributes of XML tags defined in the 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

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

Table 7.1: LLM prompts

Prompt $p0$ for $ComplianceKG_X$ generation	
1	*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)	
2	*Identify the XML tags used to configure the appropriate third-party service.*
GraphRAG retrieval: prompt $p2$ (for GraphDB)	
3	*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 $p3$ for TPS and data types retrieval	
4	*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	
5	*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:* $\{\langle SensitiveCategory_1, \{DataTypes_1 \dots DataTypes_n\}\rangle, \dots, \langle SensitiveCategory_{14}, \{DataTypes_1 \dots DataTypes_m\}\rangle\}$*.*

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.

7.1.2 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 7.1, 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 TPS but also its child tags. This is important because manifests are heterogeneous, so developers can configure TPS 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 2, 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 7.1, is sent to LLM via the *graph.query()*[10] 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 TPS.

Finally, we send the final prompt (4th row in Table 7.1) to LLM to determine the list of TPS the app integrates with and the types of data the app shares.

7.2 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 vary depending on the format, communication protocol, and regulations of the app developer.[11]

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,

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

[11] 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).

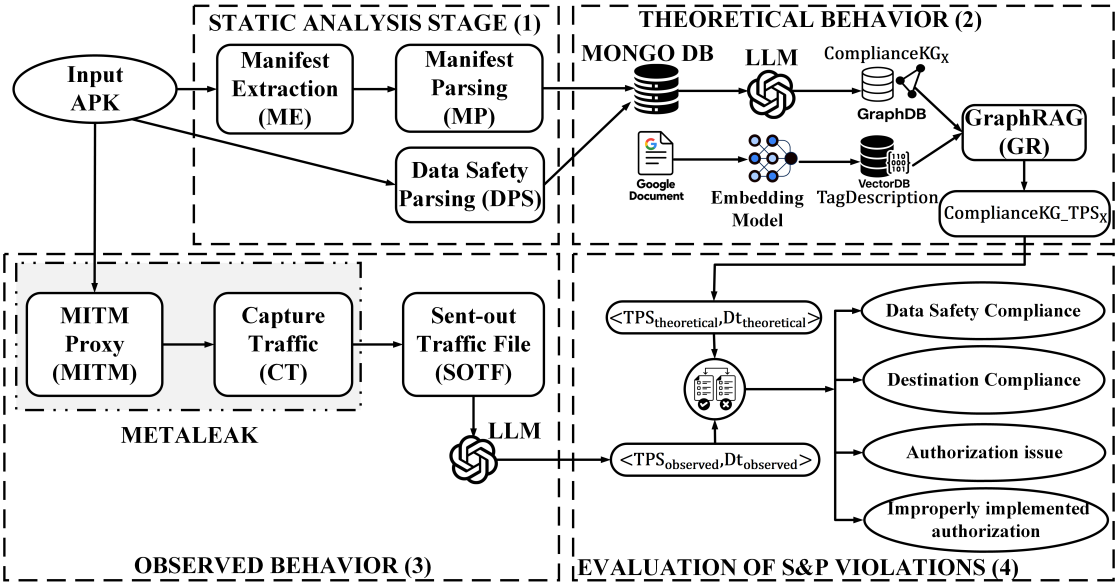

Figure 7.2: WearLeak Architecture

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 chapter, 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. Based on Table 2.1, we have 14 sensitive data categories, modeled as pairs $\langle \text{SensitiveCategory}, \text{DataTypes} \rangle$.

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 7.1, 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.

7.3 WearLeak Architecture

WearLeak is designed to receive an APK of the app X as input and return results that include (1) data safety compliance, (2) destination compliance, (3) improperly implemented authorization, and (4) the user's credential handling. It consists of four stages (cf. Figure 7.2), with all modules in each stage programmed in Python. In the following, we detail the implementation of each stage.

7.3.1 Static analysis stage

The static analysis stage aims to extract all manifest XML tags of app X. First, we build the Manifest Extraction (ME) module based on Androguard[12] to extract the Manifest.xml file from the app's APK without requiring the APK file to be decompiled. After that, the Manifest Parsing (MP) module extracts the XML tags and their attributes. Then, we use the Selenium[13] framework in the Data Safety Parsing (DSP) module to collect safety data from the Google Play Store website corresponding to the app. Finally, the tag information (i.e., name and attributes) and data safety are stored in the MongoDB database.

7.3.2 Theoretical Behavior Stage

The goal of this stage is to create ComplianceKG_TPS$_X$ of app X. The methodology of applying GraphRAG has been described in detail in Section 7.1. Specifically, we retrieve the Manifest tag and data safety information (i.e., the output of the static analysis stage) from MongoDB to assign values for the prompt $p0$ in Table 7.1 for generation ComplianceKG$_X$ (i.e., GraphDB).

Then, we use Selenium to collect the syntax, structure, and attributes of each tag t_i used in the Manifest file from Google Document to build the tuple $TagDescriptor_{t_i} = \langle syntax_{t_i}, contained\_in_{t_i}, description_{t_i}, attributes_{t_i} \rangle$. After that, we use OpenAPI's default embedding model[14] to create VectorDB from $TagDescriptor$.

Next, we perform hybrid retrieval as described in Section 7.1.2 by using prompt $p1$ for VectorDB and prompt $p2$ for GraphDB respectively as described in Table 7.1 to obtain a ComplianceKG_TPS$_X$.

Finally, we use prompt $p3$ as described in Table 7.1 to get the list of TPS the app integrates with and the types of data the app shares (i.e., $\langle TPS_{\text{theoretical}}, Dt_{\text{theoretical}} \rangle$).

7.3.3 Observed Behavior Stage

The goal of this stage is to identify the types of sensitive data that the app will actually share and the destinations of these data (i.e., $\langle TPS_{\text{observed}}, Dt_{\text{observed}} \rangle$). Specifically, we first install and run the apps under investigation on a rooted phone. To capture

[12] https://github.com/androguard/androguard
[13] https://www.selenium.dev/
[14] https://platform.openai.com/docs/guides/embeddings

the traffic sent by apps, which is essential for modeling their observed behaviors, we leverage the MetaLeak framework [52]. 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 7.4).

Finally, we insert the sent-out traffic payload and http header from SOTF into the respective $\{\text{traffic}_{\text{sent-out}}\}$ and $\{\text{http}_{\text{header}}\}$ variables of the prompt in row 5th of Table 7.1 along with the $\langle \text{SensitiveCategory}, \text{DataTypes} \rangle$ information from Table 2.1 (cf. Section 7.2). Then, the prompt is sent to LLM to determine the list of sensitive data sent and their destinations.

7.3.4 Evaluation of Security & Privacy Stage

In this stage, we compare $\langle TPS_{\text{theoretical}}, Dt_{\text{theoretical}} \rangle$ and $\langle TPS_{\text{observed}}, Dt_{\text{observed}} \rangle$ to evaluate (1) data safety compliance and (2) destination compliance. Then, we use the predefined information for registration/login as keywords to search for the existence of user's credential in plaintext in SOTF. For example, we use password = *"Password@11235"* as keyword and search SOTF, if we find the string *"Password@11235"*, it means the app sent the user's password in plaintext. Finally, we used the keywords *"token"* and *"access_token"* to identify the access token used in the authorization mechanism of apps that access health data indirectly (cf. Section 2.2). Since access tokens typically use the JWT token format[16], we use a simple Python script to parse the access token and extract the expiration time, thereby assessing whether the authorization mechanism is properly implemented (i.e., whether the token had an excessively long expiration time).

7.4 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-

[15] https://github.com/honeynet/droidbot
[16] https://www.ibm.com/docs/en/cics-ts/6.1.0?topic=cics-json-web-token-jwt

compliant apps. We first introduce the dataset used.

7.4.1 Dataset

We downloaded 5,000 apps supporting smartwatches from the Google Play Store in Europe between January and March 2025. Among these, we select the 1,000 most popular based on their installation counts. Then, we install and run each of them on a smartphone (model Samsung Galaxy M51) and examine 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 obtain a set of 711 companion apps, denoted as $\mathcal{D}_{\text{evaluation}}$. For each of these apps, we collect the corresponding Manifest and Data Safety.

7.4.2 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. 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.

7.4.2.1 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 the LLM and verify that the two lists are identical 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 TPS integrated with apps in $\text{Model}_{\text{val}}$. Specifically, we rely on the XML tags in the Google documentation used to configure TPS (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) $| TPS\_manual_X - TPS\_LLM_X | > 0$, LLM misses some of the TPS manually identified; finally, if (3) $| TPS\_LLM_X - TPS\_manual_X | > 0$, LLM has misclassified some TPS. 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 TPS, 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.

7.4.2.2 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.[17] 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 7.2 (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.[18] 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 7.1). Similarly, we identify the corresponding owners (denoted as $\text{Destination\_URL}_{\text{LLM}}$) and

[17] 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.

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

Table 7.2: 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%)

compare them with Destination_URL$_{\text{manual}}$. We find that Destination_URL$_{\text{LLM}}$ = Destination_URL$_{\text{manual}}$, for all the analyzed apps, meaning that LLM can detect 100% of the destinations from the HTTP header.

7.4.3 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 7.2 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.

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 TPS 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 TPS 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 TPS.

Apps not violating Data Safety (231 apps). 227/231 (≈98.27%) of the apps send

Table 7.3: Distribution of Access Token Expiration Times

Expiration time (t)	$t \leq 24h$	$24h < t \leq 72h$	$72h < t \leq 120h$	$t > 120h$
Quantity ($apps$)	112	87	59	29

data only to TPS 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 TPS 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) Improperly implemented authorization. We check the length of the expiration time and get the results as shown in Table 7.3. Although the apps have different expiration time configurations, none of them are equipped with a mechanism to revoke access tokens automatically when users uninstall the app. This raises concerns about the possibility of developers continuing to access health data without users' awareness.

4) 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.[19] Furthermore, the failure to encrypt users' passwords also violates GDPR Article 32 – security of processing.[20]

[19]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

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

Chapter 8

Conclusion and Future Work

It is evident that smartphones and wearable devices have asserted their important role in modern life. Global statistics project that the number of app downloads will reach $299 billion by the end of 2025 and is expected to climb to $350 billion by 2027[1]. Based on these statistics and the revenue models of the mobile and wearable ecosystems, which are primarily based on advertising instead of paid apps, it is clear that users will continue to face privacy violations, while existing regulations, such as GDPR and CCPA, are insufficient to fully protect them.

Therefore, this thesis investigates privacy compliance in mobile and wearable ecosystems, with a particular focus on how sensitive data is collected, processed, and shared under the GDPR and similar regulations. Our research follows two main directions: traditional analysis and LLM-based analysis.

In the first direction, we introduce a novel and sophisticated attack vector that can easily trick users into leaking sensitive information through EXIF metadata, leveraging their common smartphone usage habit of taking and sharing images online. Moreover, this attack vector is feasible because it exploits security vulnerabilities within the protection mechanisms provided by the Android OS. Through the MetaLeak framework (Chapter 5), we empirically demonstrate the prevalence and impact of this issue, showing that over one-fifth of 5,000 popular apps leak at least one sensitive metadata type, such as datetime, smartphone model, smartphone brand, serial number, and GPS. However, MetaLeak is a semi-automated process that requires human intervention in dynamic analysis and is therefore not scalable, a weakness inherent to traditional analysis.

To address the limitations of traditional analysis, in the second direction, we propose ALIBIS (Chapter 6), a fully automated LLM-based framework that accurately summarizes EXIF-related code and provides early warnings of potential privacy risks, achieving an average precision of 0.8902 based on the labeled dataset obtained from MetaLeak's result. We continue to extend the LLM-based solution for analyzing wearable ecosystems by developing WearLeak (Chapter 7), an LLM and Knowledge Graph to assess privacy non-compliance related to sensitive data sharing and data destinations. By analyzing 1,000 popular companion apps, our study reveals that 67.5% share sensitive data without

[1] https://www.tekrevol.com/blogs/mobile-app-download-statistics/

declaring in the Data Safety section, while 4.8% transmit data to undeclared third-party services via hidden channels, such as deep links.

Overall, the results of this thesis demonstrate that LLM-based solutions are a promising solution to support and gradually replace traditional analysis in inspecting for privacy compliance. Moreover, the proposed frameworks, MetaLeak, ALIBIS, and WearLeak, can collaborate to provide a scalable and automated foundation for detecting privacy violations across mobile and wearable ecosystems.

In this thesis, we develop methods to assess and detect privacy violations in mobile and wearable apps. However, the contributions so far primarily take the app-centric perspective, treating the app as the main actor responsible for privacy violations. This is a valid point, but incomplete, if we ignore the role of end users, who directly use the app and grant dangerous-level permissions, but often lack security/privacy knowledge. Specifically, the permission model is the primary mechanism for protecting user security and privacy; however, it still has two major weaknesses. Firstly, Android permissions are rigid as they follow a binary mechanism (yes or no), meaning that when the user has granted permission, the app continuously collects and shares the user's data. This is considered privacy-compliant. However, in reality, user behavior and preferences are extremely complex, so Android permissions, which operate according to a binary mechanism, cannot meet the user's privacy requirements (i.e., privacy preferences). For example, the user agrees to let the health app collect health data, including heart rate and steps. However, the user wants the heart rate to be collected throughout the day (i.e., stored locally) but only shared between 0:00 and 8:00 a.m. (sleep time), so that doctors can monitor the risk of cardiac arrest during sleep. In contrast, step data can only be collected and shared between 8:00 and 18:00 (for work and outdoor activities) and not collected or shared outside of this time frame. With complex privacy preferences like the example, the permission model cannot be met automatically. Of course, it can be explained that users can turn permissions on and off according to their preferred time frame, but this will be annoying and inconvenient. Secondly, permissions have a different meaning even though they affect the same type of sensitive data. For example, with the same permission to collect location data, Android provides two permissions: ACCESS_COARSE_LOCATION[2] allows the app to get the device's relative location (with an error of 700-1000m), while ACCESS_FINE_LOCATION[3] allows the app to get the device's exact location (with an error of 2-3m). The distinction is only published in the Google document without any mechanism to explain to users during the permission-granting process. Moreover, permissions are pre-configured in the `Manifest.xml` file by the app developer, so users cannot replace or revise them. Therefore, in the first future work, we plan to develop Agentic AI-based[4] solutions that support end-users in making

[2] https://developer.android.com/reference/android/Manifest.permission#ACCESS_COARSE_LOCATION

[3] https://developer.android.com/reference/android/Manifest.permission#ACCESS_FINE_LOCATION

[4] Agentic AI is a complete system comprising multiple AI-Agents working together to achieve a complex goal. AI-Agents can communicate with each other, divide tasks, plan actions, use multiple tools, and and maintain long-term memory for better reasoning [1].

informed decisions when granting permissions, and to enhance the permission model with greater flexibility so that it better reflects and respects users' privacy preferences.

In the second direction of future work, we will focus on optimizing the proposed LLM-based solutions (i.e., code summarization). Specifically, although LLM have shown great potential, there is no generalized plug-and-play solution for LLMs. For example, ALIBIS requires an understanding of how EXIF metadata is handled in the Android OS and optimizing LLM input prompts, thus requiring expert knowledge in both Android and security. This leads to LLMs not yet being a widely accessible solution for the general public to independently apply LLMs to address security and privacy risks. Besides, most of the state-of-the-art researches leverage LLMs to perform code summarization to understand how apps handle sensitive data, which is an appropriate approach but still insufficient. Specifically, the source code may contain code blocks that remove sensitive information from data shared with third parties; however, there is no guarantee that these code blocks are executed at runtime. Hackers who understand how tools use code summarization to analyze app behavior can pretend to implement privacy-compliant code blocks but do not actually execute them. Therefore, in the future, code summarization should be followed by code execution to verify the results. Code execution is supported by the GPT-4 model and its variants (cf. Table 2.2) but is not yet widely adopted.

Lastly, we will examine the security and privacy issues associated with the LLM itself and the systems built upon it. First, LLMs operate as a black box, so it is difficult to fully understand the entire architecture and the data used to train models. Specifically, Zilong Lin et al. [42] found that unverified LLMs can be packaged into malicious services. Currently, numerous LLMs, including malicious services, are being introduced on the black market, primarily offering features such as malicious code generation, phishing emails, and scam sites, for example, DarkGPT[5] and BadGPT[6]. These features are very popular because they support hackers in attacking users without requiring as much knowledge as before. It is also not excluded that these malicious LLMs steal information from the people who use them—a double-edged sword. Therefore, it is recommended to use verified LLMs. Second, although retraining LLMs is almost impossible, hackers can still manipulate the responses of LLMs by attacking the context-enhanced processes through content poisoning attacks. Figure 8.1 describes content poisoning attacks on the RAG workflow. Specifically, hackers can create a malicious database and mix it with a benign database to control the responses of LLMs. Suppose a simple example is as follows:

User prompt: *Please guide me on how to store API keys securely to avoid the risks of OSWAP R1 - improper credential usage.*

LLM response: *You can store the API key in Amazon S3 storage at https://hacker-bucket-name.s3.amazonaws.com/uploads/ and access your key dynamically when using it.*

In this example, if the user completely trusts the answers provided by the manipu-

[5] https://github.com/codewithdark-git/DarkGPT
[6] https://flowgpt.com/p/badgpt-1

lated LLMs, their API key will be stolen.

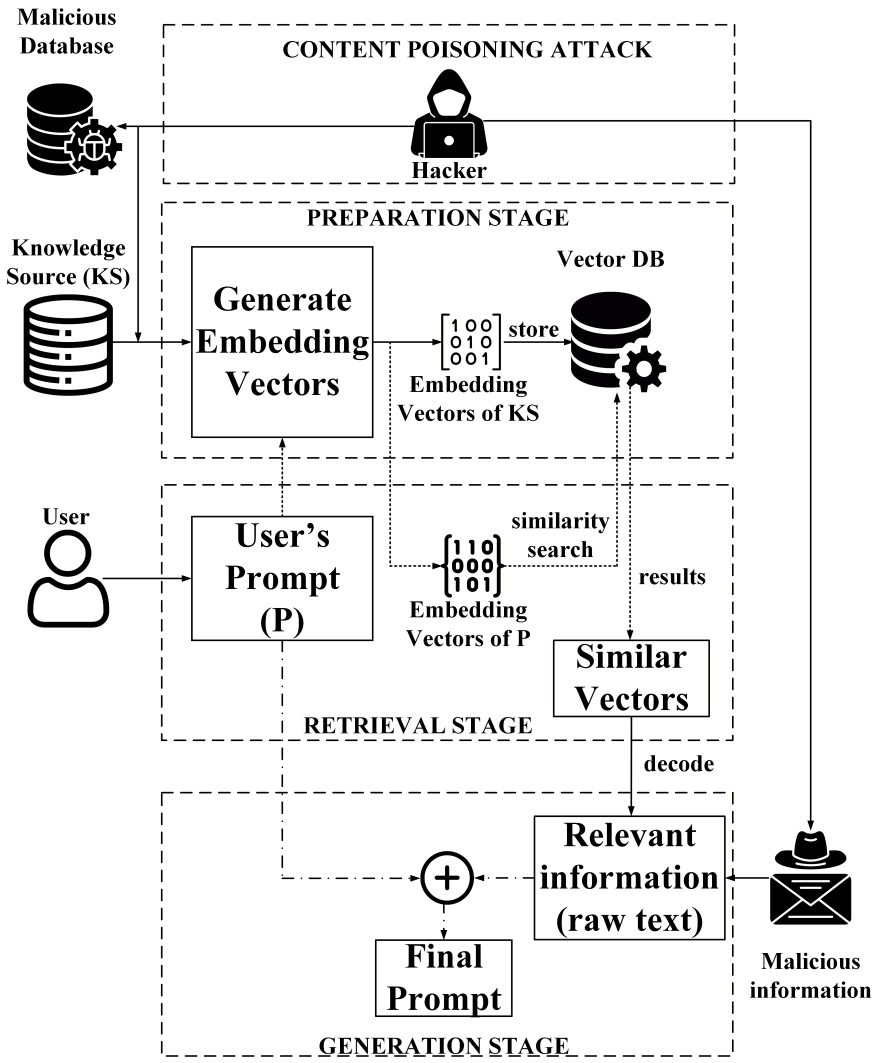

Figure 8.1: Content poisoning attacks on the RAG workflow

Finally, a potential research direction is to develop a system that helps developers ensure privacy compliance throughout the software development process, rather than only detecting violations after the application has been released. The idea is to build a *privacy-by-design* architecture based on LLM, in which LLM is combined with a RAG mechanism using embedded vector database clusters (VectorDB cluster) for various purposes. First, VectorDB-1 can be constructed from OWASP Mobile Top 10 resources, including vulnerability descriptions, insecure code examples, and API-misuse patterns commonly associated with data leakage. This enables LLM to retrieve and compare the code that the programmer is writing with the risky code patterns identified by OWASP,

thereby providing early warnings and suggesting fixes that adhere to security standards. Second, a vector database (VectorDB-2) includes information about popular SDKs (e.g., advertising, analytics, crash-reporting, social login), along with instructions for enabling or disabling (opt-in/opt-out) data collection. When developers integrate third-party SDKs, LLM can verify whether the implementation is privacy-compliant, suggest mechanisms to limit sensitive data collection, and identify cases where SDKs collect data contrary to user expectations. This approach helps reduce privacy violations caused by unintentional developer mistakes and encourages the creation of mobile and wearable applications that adhere to privacy-by-design principles from the beginning.

Bibliography

[1] Deepak Bhaskar Acharya, Karthigeyan Kuppan, and B Divya. Agentic ai: Autonomous intelligence for complex goals–a comprehensive survey. *IEEe Access*, 2025.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*, pages 1–5, 2022.

[4] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[5] Lin Ai, Ziwei Gong, Harshsaiprasad Deshpande, Alexander Johnson, Emmy Phung, Ahmad Emami, and Julia Hirschberg. Novascore: A new automated metric for evaluating document level novelty. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 3479–3494, 2025.

[6] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th international conference on mining software repositories*, pages 468–471, 2016.

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[8] Haya Altuwaijri and Sanaa Ghouzali. Android data storage security: A review. *Computer and Information Sciences*, 32(5):543–552, 2020.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices*, 49(6):259–269, 2014.

[10] Wenying Bao, Wenbin Yao, Ming Zong, and Dongbin Wang. Cross-site scripting attacks on android hybrid applications. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pages 56–61, 2017.

[11] Rajiv Chandawarkar and Prakash Nadkarni. Safe clinical photography: best practice guidelines for risk management and mitigation. *Archives of Plastic Surgery*, 48(03):295–304, 2021.

[12] 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 *International Conference on Science of Cyber Security*, pages 541–556. Springer, 2022.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[14] Dhruv Dhamani and Mary Lou Maher. The tyranny of possibilities in the design of task-oriented llm systems: A scoping survey. *arXiv preprint arXiv:2312.17601*, 2023.

[15] Zikan Dong, Tianming Liu, Jiapeng Deng, Li Li, Minghui Yang, Meng Wang, Guosheng Xu, and Guoai Xu. Exploring covert third-party identifiers through external storage in the android new era. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4535–4552, 2024.

[16] Xiaoyu Du and Mark Scanlon. Methodology for the automated metadata-based classification of incriminating digital forensic artefacts. In *Proceedings of the 14th international conference on availability, reliability and security*, pages 1–8, 2019.

[17] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[18] Jeremy Faircloth. Chapter 8 - client-side attacks and social engineering. In Jeremy Faircloth, editor, *Penetration Tester's Open Source Toolkit (Fourth Edition)*, pages 273–318. Syngress, Boston, fourth edition edition, 2017.

[19] Wenhao Fan, Daishuai Zhang, Ye Chen, Fan Wu, and Yuan'an Liu. Estidroid: estimate api calls of android applications using static analysis technology. *IEEE Access*, 8:105384–105398, 2020.

[20] Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. Llm agents can autonomously exploit one-day vulnerabilities. *arXiv preprint arXiv:2404.08144*, 2024.

[21] Sidong Feng and Chunyang Chen. Prompting is all you need: Automated android bug replay with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[23] Pascal Gadient, Marc-Andrea Tarnutzer, Oscar Nierstrasz, and Mohammad Ghafari. Security smells pervade mobile app servers. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2021.

[24] Shivi Garg and Niyati Baliyan. Comparative analysis of android and ios from security viewpoint. *Computer Science Review*, 40:100372, 2021.

[25] Kushankur Ghosh, Colin Bellinger, Roberto Corizzo, Paula Branco, Bartosz Krawczyk, and Nathalie Japkowicz. The class imbalance problem in deep learning. *Machine Learning*, 113(7):4845–4901, 2024.

[26] Charles Gouert and Nektarios Georgios Tsoutsos. Dirty metadata: Understanding a threat to online privacy. *IEEE Security & Privacy*, 20(6):27–34, 2022.

[27] Moritz Gruber, Christian Höfig, Maximilian Golla, Tobias Urban, and Matteo Große-Kampmann. "we may share the number of diaper changes": A privacy and security analysis of mobile child care applications. *Proceedings on Privacy Enhancing Technologies*, 3:394–414, 2022.

[28] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[29] Maanak Gupta, CharanKumar Akiri, Kshitiz Aryal, Eli Parker, and Lopamudra Praharaj. From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy. *IEEE Access*, 2023.

[30] Haoyu Han, Harry Shomer, Yu Wang, Yongjia Lei, Kai Guo, Zhigang Hua, Bo Long, Hui Liu, and Jiliang Tang. Rag vs. graphrag: A systematic evaluation and key insights. *arXiv preprint arXiv:2502.11371*, 2025.

[31] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A Rossi, Subhabrata Mukherjee, Xianfeng Tang, et al. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309*, 2024.

[32] Benjamin Henne, Maximilian Koch, and Matthew Smith. On the awareness, control and privacy of shared photo metadata. In *International Conference on Financial Cryptography and Data Security*, pages 77–88. Springer, 2014.

[33] Han Hu, Han Wang, Ruiqi Dong, Xiao Chen, and Chunyang Chen. Enhancing gui exploration coverage of android apps with deep link-integrated monkey. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–31, 2024.

[34] Yujin Huang and Chunyang Chen. Smart app attack: hacking deep learning models in android apps. *IEEE Transactions on Information Forensics and Security*, 17:1827–1840, 2022.

[35] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research*, 24(251):1–43, 2023.

[36] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE, 2023.

[37] Vasileios Kouliaridis, Georgios Karopoulos, and Georgios Kambourakis. Assessing the effectiveness of llms in android application vulnerability analysis. *arXiv preprint arXiv:2406.18894*, 2024.

[38] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 183–196, 2017.

[39] Fenghua Li, Xinyu Wang, Ben Niu, Hui Li, Chao Li, and Lihua Chen. Exploiting location-related behaviors without the gps data on smartphones. *Information Sciences*, 527:444–459, 2020.

[40] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017.

[41] Zhen Li, Gang Xiong, and Li Guo. Unveiling ssl/tls mitm hosts in the wild. In *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pages 141–145. IEEE, 2020.

[42] Zilong Lin, Jian Cui, Xiaojing Liao, and XiaoFeng Wang. Malla: Demystifying real-world large language model integrated malicious services. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024.

[43] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. Chatting with gpt-3 for zero-shot human-like mobile automated gui testing. *arXiv preprint arXiv:2305.09434*, 2023.

[44] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[45] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Zhilin Tian, Yuekai Huang, Jun Hu, and Qing Wang. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[46] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. Owl eyes: Spotting ui display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 398–409, 2020.

[47] Qian Luo, Yinbo Yu, Jiajia Liu, and Abderrahim Benslimane. Automatic detection for privacy violations in android applications. *IEEE Internet of Things Journal*, 9(8):6159–6172, 2021.

[48] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.

[49] Gabriel Morales, KC Pragyan, Sadia Jahan, Mitra Bokaei Hosseini, and Rocky Slavin. A large language model approach to code and privacy policy alignment. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 79–90. IEEE, 2024.

[50] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*, 2023.

[51] Lam Tran Thanh Nguyen, Son Xuan Ha, Trieu Hai Le, Huong Hoang Luong, Khanh Hong Vo, Khoi Huynh Tuan Nguyen, Tuan Anh Dao, Hy Vuong Khang Nguyen, et al. Bmdd: a novel approach for iot platform (broker-less and microservice architecture, decentralized identity, and dynamic transmission messages). *PeerJ Computer Science*, 8:e950, 2022.

[52] Tran Thanh Lam Nguyen, Barbara Carminati, and Elena Ferrari. Metaleak: Assessing image metadata leakage in android apps. In *2024 IEEE/ACS 21st International Conference on Computer Systems and Applications (AICCSA)*, pages 1–10, 2024.

[53] Tran Thanh Lam Nguyen, Barbara Carminati, and Elena Ferrari. Llms on support of privacy and security of mobile apps: state of the art and research directions. *arXiv preprint arXiv:2506.11679*, 2025.

[54] Trung Tin Nguyen, Michael Backes, and Ben Stock. Freely given consent? studying consent notice of third-party tracking and its violations of gdpr in android apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2369–2383, 2022.

[55] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. A review on the attention mechanism of deep learning. *Neurocomputing*, 452:48–62, 2021.

[56] Sahrima Jannat Oishwee, Natalia Stakhanova, and Zadia Codabux. Large language model vs. stack overflow in addressing android permission related challenges. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 373–383. IEEE, 2024.

[57] Babatunde Olabenjo and Dwight Makaroff. Information leakage in wearable applications. In *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*, pages 211–224. Springer, 2019.

[58] Shirui Pan, Linhao Luo, Yufei Wang, Chen Chen, Jiapu Wang, and Xindong Wu. Unifying large language models and knowledge graphs: A roadmap. *IEEE Transactions on Knowledge and Data Engineering*, 36(7):3580–3599, 2024.

[59] 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*, pages 2445–2458, 2022.

[60] Rishank Pratik and R Sendhil. Privacy protection against reverse image search. In *2023 Third International Conference on Artificial Intelligence and Smart Energy (ICAIS)*, pages 1207–1214. IEEE, 2023.

[61] Bambang Purnomosidi Dwi Putranto, Robertus Saptoto, Ovandry Chandra Jakaria, and Widyastuti Andriyani. A comparative study of java and kotlin for android mobile application development. In *2020 3rd International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, pages 383–388. IEEE, 2020.

[62] 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)*, pages 603–620, 2019.

[63] David Rodriguez, Joseph A Calandrino, Jose M Del Alamo, and Norman Sadeh. Privacy settings of third-party libraries in android apps: A study of facebook sdks. *Proceedings on Privacy Enhancing Technologies*, 2025.

[64] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[65] Antonio Ruggia, Eleonora Losiouk, Luca Verderame, Mauro Conti, and Alessio Merlo. Repack me if you can: An anti-repackaging solution based on android virtualization. In *Proceedings of the 37th Annual Computer Security Applications Conference*, pages 970–981, 2021.

[66] Kunal Sawarkar, Abhilasha Mangal, and Shivam Raj Solanki. Blended rag: Improving rag (retriever-augmented generation) accuracy with semantic search and hybrid query-based retrievers. *arXiv preprint arXiv:2404.07220*, 2024.

[67] Tianhao Shen, Renren Jin, Yufei Huang, Chuang Liu, Weilong Dong, Zishan Guo, Xinwei Wu, Yan Liu, and Deyi Xiong. Large language model alignment: A survey. *arXiv preprint arXiv:2309.15025*, 2023.

[68] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[69] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt engineering or fine-tuning: An empirical assessment of llms for code. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 490–502. IEEE, 2025.

[70] Ha Xuan Son, Barbara Carminati, and Elena Ferrari. A risk assessment mechanism for android apps. In *2021 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pages 237–244. IEEE, 2021.

[71] Ha Xuan Son, Barbara Carminati, and Elena Ferrari. A risk estimation mechanism for android apps based on hybrid analysis. *Data Science and Engineering*, 7(3):242–252, 2022.

[72] Yisheng Song, Ting Wang, Puyu Cai, Subrota K Mondal, and Jyoti Prakash Sahoo. A comprehensive survey of few-shot learning: Evolution, applications, challenges, and opportunities. *ACM Computing Surveys*, 55(13s):1–40, 2023.

[73] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE communications surveys & tutorials*, 20(1):465–488, 2017.

[74] Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. Scandroid: Automated side-channel analysis of android apis. In *Proceedings of the 11th acm conference on security & privacy in wireless and mobile networks*, pages 224–235, 2018.

[75] Mohammad Tahaei, Alisa Frik, and Kami Vaniea. Deciding on personalized ads: Nudging developers about user privacy. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 573–596, 2021.

[76] Zeya Tan and Wei Song. Ptpdroid: Detecting violated user privacy disclosures to third-parties of android apps. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 473–485, 2023.

[77] Shahab Tayeb, Abigail Week, Joshua Yee, Mayra Carrera, Kuira Edwards, Vicki Murray-Garcia, Meghann Marchello, Justin Zhan, and Matin Pirouz. Toward metadata removal to preserve privacy of social media users. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 287–293. IEEE, 2018.

[78] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[79] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[80] Imdad Ullah, Roksana Boreli, and Salil S Kanhere. Privacy in targeted advertising on mobile devices: a survey. *International Journal of Information Security*, 22(3):647–678, 2023.

[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[82] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, pages 382–394, 2022.

[83] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st annual computer security applications conference*, pages 61–70, 2015.

[84] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)*, 53(3):1–34, 2020.

[85] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Transactions on Knowledge Discovery from Data*, 18(7):1–34, 2024.

[86] Haohuang Wen, Juanru Li, Yuanyuan Zhang, and Dawu Gu. An empirical study of sdk credential misuse in ios apps. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 258–267. IEEE, 2018.

[87] Pascal Wichmann, Alexander Groddeck, and Hannes Federrath. Fileuploadchecker: detecting and sanitizing malicious file uploads in web applications at the request level. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–10, 2022.

[88] Shangyu Wu, Ying Xiong, Yufei Cui, Haolun Wu, Can Chen, Ye Yuan, Lianming Huang, Xue Liu, Tei-Wei Kuo, Nan Guan, et al. Retrieval-augmented generation for natural language processing: A survey. *arXiv preprint arXiv:2407.13193*, 2024.

[89] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):1–41, 2011.

[90] Doguhan Yeke, Muhammad Ibrahim, Güliz Seray Tuncay, Habiba Farrukh, Abdullah Imran, Antonio Bianchi, and Z. Berkay Celik. Wear's my data? understanding the cross-device runtime permission model in wearables. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2404–2421, 2024.

[91] Yagmur Yigit, William J Buchanan, Madjid G Tehrani, and Leandros Maglaras. Review of generative ai methods in cybersecurity. *arXiv preprint arXiv:2403.08701*, 2024.

[92] Kanae Yoshida, Hironori Imai, Nana Serizawa, Tatsuya Mori, and Akira Kanaoka. Understanding the origins of weak cryptographic algorithms used for signing android apps. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 713–718, 2018.

[93] Chaoning Zhang, Chenshuang Zhang, Sheng Zheng, Yu Qiao, Chenghao Li, Mengchun Zhang, Sumit Kumar Dam, Chu Myaet Thwal, Ye Lin Tun, Le Luang Huy, et al. A complete survey on generative ai (aigc): Is chatgpt from gpt-4 to gpt-5 all you need? *arXiv preprint arXiv:2303.11717*, 2023.

[94] Hao Zhang, Zhuolin Li, Hossain Shahriar, Dan Lo, Fan Wu, and Ying Qian. Protecting data in android external data storage. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 924–925, 2019.

[95] Wenxiang Zhao, Juntao Wu, and Zhaoyi Meng. Apppoet: Large language model based android malware detection via multi-view prompt engineering. *Expert Systems with Applications*, 262:125546, 2025.

[96] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM conference on security & privacy in wireless and mobile networks*, pages 1–12, 2015.

Appendices

Appendix A

User–Developer Awareness Survey on EXIF Metadata

To assess the awareness of security risks associated with EXIF metadata, we conduct a survey. The survey was designed as a structured questionnaire using the Google Forms platform (cf. Section A.1). This survey combines both single-choice and multiple-choice prepared questions. The questionnaire was developed based on previous literature on EXIF metadata security risks, including: attack types [26, 77] targeting the five sensitive metadata types that are the focus of this study, and user awareness of EXIF [32]. Specifically, based on prior studies of EXIF-related attacks, the questionnaire asked respondents if they were aware of risks such as geolocation tracking (i.e., social engineering attack) or device identification (re-identification attack). In addition, informed by earlier work on user awareness of EXIF metadata, we included questions about how familiar respondents were with EXIF data, whether they had ever attempted to remove it, and which types of metadata they perceived as the most sensitive. Survey participation was completely voluntary and anonymous, with no personally identifiable information collected unless voluntarily provided for future participation. The survey complied with GDPR, and the collected data was only stored for 30 days.

The survey targeted two user groups: (1) Android app developers and (2) Android app users. We designed distinct questionnaires for Android app users and developers. The survey was conducted over one week with the participation of 130 individuals from various countries, including Italy, Vietnam, Germany, South Korea, the UK, Australia, and the USA. Additionally, for the developer role, we gathered information from three companies in India, Vietnam, and the Czech Republic, as well as from freelancers. Specifically, the participants comprised 77 Android app users (59.2%) and 53 Android app developers (40.8%). Finally, the survey results are presented in Section A.2.

A.1 Survey Questionnaire

In this section, we present two separate survey questionnaires designed for two user groups. Each group is only required to fill out the questionnaire relevant to their role

(i.e., developer and/or user).[1] However, if a participant is both an app developer and an app user, they are requested to complete both questionnaires.

A.1.1 Questions for participants who choose the developer role

1. Are you aware that images can contain metadata (EXIF data)?

- ◯ Yes
- ◯ No
- ◯ Not sure

2. Which of the following information do you think EXIF metadata can contain? (Select all that apply)

- ☐ Date and time the photo was taken
- ☐ GPS coordinates of where the photo was taken
- ☐ Camera make and model
- ☐ Device's unique identifier
- ☐ None of the above

3. Do you believe that leaking EXIF metadata can pose a privacy risk?

- ◯ Yes, definitely
- ◯ Possibly
- ◯ Not sure
- ◯ No, it's harmless

4. Which of the following do you consider potential risks associated with leaked EXIF metadata? (Select all that apply)

- ☐ Revealing the location where the photo was taken
- ☐ Exposing the date and time of the photo
- ☐ Disclosing the camera make and model
- ☐ Compromising personal or organizational security

[1] This setup is suitable for cases where a user only develops Android apps but personally chooses an iOS device.

□ None of the above

5. How often do you consider the presence of EXIF metadata in images when developing apps?

◯ Always

◯ Often

◯ Sometimes

◯ Rarely

◯ Never

6. How often do you consider removing EXIF metadata in images when developing apps?

◯ Always

◯ Often

◯ Sometimes

◯ Rarely

◯ Never

7. Do you use any libraries or tools to manage or remove EXIF metadata in your applications?

◯ Yes, regularly

◯ Yes, occasionally

◯ No, but I am aware of such tools

◯ No, I am not aware of such tools

8. How important do you think it is to educate developers about the risks of EXIF metadata?

◯ Extremely important

◯ Important

◯ Somewhat important

◯ Not important

◯ Irrelevant

9. Would you consider implementing metadata removal in your development process if it was easy to integrate?

◯ Yes, definitely

◯ Probably

◯ Maybe

◯ Unlikely

◯ No, not interested

10. What do you think are the best practices for managing EXIF metadata in images within applications? (Select all that apply)

☐ Removing all metadata by default

☐ Allowing users to choose which metadata they want to remove

☐ Keep metadata as they are

A.1.2 Questions for participants who choose the user role

1. How often do you take photos with your smartphone?

◯ Multiple times a day

◯ Daily

◯ Weekly

◯ Monthly

◯ Rarely/Never

2. How often do you share your photos online (e.g., social media, messaging apps)?

◯ Multiple times a day

◯ Daily

◯ Weekly

◯ Monthly

◯ Rarely/Never

3. Are you aware that images you take with your smartphone contain metadata (EXIF data) that includes sensitive information?

- ◯ Yes, I am fully aware
- ◯ Yes, but I am not sure what kind of information it includes
- ◯ No, I was not aware
- ◯ I don't know what is EXIF metadata

4. Which of the following information do you think EXIF metadata can contain? (Select all that apply)

- ☐ Date and time the photo was taken
- ☐ GPS coordinates of where the photo was taken
- ☐ Camera make and model
- ☐ Device's unique identifier
- ☐ I don't know

5. Do you believe that EXIF metadata in photos can pose a privacy or security risk if leaked?

- ◯ Yes, definitely
- ◯ Possibly
- ◯ Not sure
- ◯ No, it's harmless

6. Have you ever experienced or heard of any incidents where leaked EXIF metadata caused problems (e.g., revealing location, compromising privacy)?

- ◯ Yes, personally experienced
- ◯ Yes, heard of it happening to others
- ◯ No, never
- ◯ I feel like there's a problem but not sure

7. How important do you think it is to educate users about the risks of EXIF metadata?

- ◯ Extremely important

- ◯ Important
- ◯ Somewhat important
- ◯ Not important
- ◯ Irrelevant

8. Do you delete or remove EXIF metadata from your photos before sharing them online?

- ◯ Always
- ◯ Often
- ◯ Sometimes
- ◯ Rarely
- ◯ Never

9. Does Android ever warn you about the existence of sensitive EXIF metadata in your photos?

- ◯ Yes
- ◯ No
- ◯ I don't know

10. Do you know how to remove EXIF metadata from your photos if you want to?

- ◯ Yes, I know how
- ◯ I have some idea, but not sure
- ◯ No, I don't know how
- ◯ I don't care about removing metadata

11. If you were provided with a tool to remove EXIF metadata when opening an image, would you be willing to use it?

- ◯ Yes, definitely
- ◯ Probably
- ◯ Maybe
- ◯ Unlikely
- ◯ No, I would keep the metadata

A.2 Survey Results

This section presents the survey results for the two target groups of our study, Android users and Android developers, to assess their awareness and practices regarding privacy risks associated with EXIF metadata. Additionally, through the survey results, we also understand that, in reality, app developers are aware of the risks related to EXIF metadata but do not proactively implement removal measures because they are not required to do so. However, most app developers indicated their willingness to adopt preventive measures if provided with efficient supporting libraries. This is the motivation for us to release the ExifMetadataLib library (cf. Section 6.4.5).

Table A.1: Survey results

Question	1	2	3	4	5
APP DEVELOPERS					
Awareness of EXIF metadata containing sensitive information	7.5%	0%	26.5%	0%	66.0%
Belief that leakage of EXIF metadata is risky	0%	0%	34.0%	39.6%	26.4%
Consider removing EXIF metadata in the app development	81.1%	3.8%	13.2%	0%	1.9%
Essential to educate developers on EXIF metadata risks	0%	0%	22.6%	71.7%	5.7%
APP USERS					
Frequency of taking images	0%	3.8%	32.3%	41.5%	22.3%
Frequency of sharing images online	0%	11.6%	32.3%	36.2%	20.0%
Awareness of EXIF metadata containing sensitive information	28.5%	23.8%	18.5%	0%	29.2%
Proactively removing EXIF metadata before sharing	78.5%	3.8%	13.8%	0%	3.8%
Android warnings about sensitive EXIF metadata	90.2%	9.8%	0%	0%	0%
Knowledge of how to remove EXIF metadata	25.4%	36.9%	0%	10.0%	27.7%
Willingness to use a tool to remove EXIF metadata	2.3%	0%	20.8%	47.7%	29.2%

Table A.1 presents the survey results. The measurement scale is divided into five levels, with level 1 representing the lowest level of agreement and level 5 representing the highest level of agreement.

For app developers, the survey results indicate that 66% of participants are aware of sensitive metadata in images, and 66% (26.4% at level 5 and 39.6% at level 4) believe that leaking sensitive metadata poses a security risk. However, only 1.9% of developers consider removing EXIF metadata during the app development. We discussed this paradox with developers and discovered that the primary cause stems from company policies or client requirements. Developers, particularly freelancers, often complete their work according to the customer's requirements. Consequently, even if they are aware of the security risks, they do not proactively implement measures to mitigate these risks unless explicitly requested by their customers.

For app users, although the frequency of taking many photos and sharing them online is 63.8% (22.3% at level 5 and 41.5% at level 4) and 56.2% (20% at level 5 and 36.2% at level 4) respectively, a substantial 70.8% (28.5% at level 1, 23.8% at level 2, and 18.5% at level 3) of users lack a clear understanding of EXIF metadata in images. Furthermore, 82.3% (78.5% at level 1 and 3.8% at level 2) of users do not actively delete sensitive metadata before sharing, whereas 62.3% (25.4% at level 1 and 36.9% at level 2) do not

know how to remove sensitive metadata from images. Encouragingly, 76.9% of users (29.2% at level 5 and 47.7% at level 4) would consider removing sensitive metadata if the app provided warnings and support for this feature.