Chapter 1

LLMs on support of privacy and security of mobile apps: state of the art and research directions

Tran Thanh Lam Nguyen,[1*] Barbara Carminati,[1*] and Elena Ferrari[1*]

[1]*Department of Theoretical and Applied Science (DISTA), University of Insubria, 21100, Varese, Italy*

*Email: ttlnguyen,barbara.carminati,elena.ferrari@uninsubria.it

Modern life has witnessed the explosion of mobile devices. However, besides the valuable features that bring convenience to end users, security and privacy risks still threaten users of mobile apps. The increasing sophistication of these threats in recent years has underscored the need for more advanced and efficient detection approaches. In this chapter, we explore the application of Large Language Models (LLMs) to identify security risks and privacy violations and mitigate them for the mobile application ecosystem. By introducing state-of-the-art research that applied LLMs to mitigate the top 10 common security risks of smartphone platforms, we highlight the feasibility and potential of LLMs to replace traditional analysis methods,

such as dynamic and hybrid analysis of mobile apps. As a representative example of LLM-based solutions, we present an approach to detect sensitive data leakage when users share images online—a common behavior of smartphone users nowadays. Finally, we discuss open research challenges.

Keywords: Mobile Apps, Privacy, LLMs, Security

1.1. Introduction

It is undeniable that mobile devices, especially smartphones, play an extremely important role in modern life and have changed people's lifestyles. Smartphones can meet most people's requirements for work, study, shopping, health care, and entertainment in a compact size. In fact, the usage of smartphones has surpassed personal computers and today dominates the digital device market. Statistics *statcounter.com (November 2024)*[1] show that mobile market share accounts for 64.04% while computers constitute 35.96%, witnessing a significant shift from computers demand to mobile devices. Due to their high mobility, smartphones bring great convenience. People can use smartphones anywhere and at any time. As a matter of fact, mobile users spend an average of 203 minutes per day viewing social media content on their devices; 70% of learners report that they feel more motivated when using mobile devices for learning, as opposed to desktop; in addition, 61% of emails are sent from mobile devices, compared to only 5% from desktop.[2].With the increasing usage of smartphones, mobile apps are gradually becoming a billion-dollar market. According to data from *Statista*, the mobile app market is expected to reach

[1]https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide
[2]https://research.com/software/mobile-vs-desktop-usage

$781.70 billion by 2029.[3]

Until August 2024, the Android operating system (OS) leads the mobile operating system market with 71.67%, while Apple iOS accounts for 27.73% and other OSs account for 0.6%.[4] As of October 2024, approximately 97% of all Android apps are free, with only 3% requiring payment.[5] These free applications have contributed to the explosion of the mobile device ecosystem, making it more attractive because users do not have to pay for using the apps' services. However, *"there is no such thing as a free lunch"*, and mobile apps earn money mainly from advertising, which accounts for 65% of total revenue.[6] These statistics show that users are the main revenue target for mobile app developers. Specifically, apps silently collect users' habits, interests, and personal information, such as posts, comments, locations, health status, and other personal data. Then, this sensitive information is shared with advertising platforms and analyzed by recommendation systems [Roy and Dutta, 2022] to deliver targeted advertising tailored to each user profile. These advertisement campaigns have the benefit of allowing users to quickly approach products and services that suit their needs. However, profile-based advertising is a double-edged sword because the accuracy of recommendations is directly proportional to the details of the user's personal information provided.

In 2018, the European Union passed the GDPR (General Data Protection Regulation),[7] which sets strict regulations on how organizations and businesses collect, process, and store EU citizens' data, such as users having the right to

[3]https://www.statista.com/outlook/amo/app/worldwide

[4]https://gs.statcounter.com/os-market-share/mobile/worldwide

[5]https://www.statista.com/statistics/266211/distribution-of-free-and-paid-android-apps/

[6]https://www.mobiloud.com/blog/mobile-app-market-statistics

[7]https://gdpr-info.eu/

be informed about the type of data collected and the right to delete personal data to protect their privacy. Specifically, apps must require user consent before collecting and sharing users' data, especially when sharing with third parties (e.g., for advertising purposes). However, GDPR is not enough to protect users comprehensively because of two main reasons: (1) most developers continue to collect users' data for profit purposes [Tahaei et al., 2021], despite GDPR, and (2) hackers take advantage of app security vulnerabilities to steal information (cf. Section 1.3). Therefore, there are significant challenges for researchers in identifying security and privacy vulnerabilities and risks of sensitive data leakage for mobile apps.

Traditionally, there are three main methods for analyzing apps' security and privacy risks, namely static analysis, dynamic analysis, and hybrid analysis. Static analysis analyzes apps' function calls, command sequences, code structure, API calls, and variables without running the app. However, static analysis does not offer a real-time observation of security or privacy violations of an app. Dynamic analysis overcomes the limitations of static analysis by focusing on identifying vulnerabilities through the app's behavior at runtime. However, implementing dynamic analysis on a large scale is complicated because this method requires tremendous resources and costs [Reardon et al., 2019]. In addition, dynamic analysis can be bypassed by apps that attempt to mimic legitimate behaviors (e.g., apps that pretend to request user consent but ignore the user's response) [Son et al., 2022]. The combination of static and dynamic analysis, called hybrid analysis, leverages the advantages of both methods. In particular, static analysis can be used initially to identify potentially risky code blocks, thus reducing the number of apps that require dynamic analysis and decreasing the computational burden that is a limitation

of dynamic analysis. Then, dynamic analysis is performed on the collection of apps returned by static analysis to observe the apps' behavior in real-time and verify the static analysis results.

There are numerous proposals in the literature employing hybrid analysis to assess the security/privacy risks of mobile apps (e.g., [Reardon et al., 2019, Lam et al., 2024]). However, these frameworks commonly have three significant weaknesses. First, the framework has to trade off accuracy and scalability. Given the vast number of apps with complex user interfaces (UIs) and interaction methods, creating a generalized automated process applicable to all apps for scripted interactions is challenging. Second, mobile apps often operate as black boxes; thus, one cannot know what data the app sends and receives. Therefore, hybrid frameworks must run on a test environment with rooted devices for reading transmitted data in and out of the app and this makes the implementation more difficult since rooting methods for each Android version are different. Third, hybrid analysis is mainly suited for identifying evidence of known security/privacy vulnerabilities.

Motivated to overcome the above mentioned limitations, this chapter discusses how to leverage large language models (LLMs) to assist in identifying sensitive information leaks in mobile apps. LLMs are powerful deep learning models pre-trained on extensive datasets, effective in tasks such as code summarization [Ahmed and Devanbu, 2022], bug detection [Wen et al., 2024], bug reproduction [Kang et al., 2023, Feng and Chen, 2024], and vulnerability exploitation [Fang et al., 2024]. Specifically, LLMs can generate automatic interactions with the app's UI following pre-defined scenarios by analyzing the app's interface. In addition, LLMs can receive feedback from the UI to infer the next interaction steps so it can be applied to many different apps while en-

suring high accuracy. Thus, LLMs are well-suited to replace dynamic analysis, a component responsible for the poor scalability of hybrid analysis (cf. Section 1.4.2). Next, by understanding app logic through the code summarization capabilities, LLMs can capture the sources and destinations of data flows inside the app and identify which sensitive information is sent out from the app without rooting the device like traditional methods (cf. Section 1.4.1). Finally, the combination of static analysis and LLMs is perfectly matched to integrate with app store software testing processes. Specifically, LLM outperforms traditional malware detection methods in detecting new malicious apps that were never included in the training dataset (cf. Section 1.4.3).

In this chapter, after introducing some background information about LLMs, we discuss how LLMs can contribute to mitigating the most notable security/privacy risks of mobile apps by reviewing state-of-the-art proposals in the field. In the chapter, we mainly focus on Android apps because they are more vulnerable to security/privacy threats than iOS [Garg and Baliyan, 2021]. Then, as a representative example of LLM usage, we present an LLMs-based architecture to detect the risk of sensitive data leakage through online image sharing, a very common habit of mobile users. Finally, we discuss open research challenges in the field.

More specifically, the remainder of this chapter is organized as follows. Section 1.2 introduces LLMs. Section 1.3 presents the primary privacy/security risks associated with mobile apps, whereas Section 1.4 describes state-of-the-art LLM-based solutions for mobile security and privacy. As an example of the usage of LLMs, Section 1.5 targets a security weakness in Android that results in the leakage of sensitive information when users share images online, along with our proposed solution utilizing LLMs to detect this risk automatically.

Section 1.6 discusses open research issues, whereas Section 1.7 concludes the chapter.

1.2. Background on LLMs

This section presents basic concepts of large language models (LLMs) [Shen et al., 2023, Minaee et al., 2024] and their applications, with a primary emphasis on security and privacy. Furthermore, we examine two important approaches employed to enhance the contextual awareness of LLMs, namely, few-shot learning (FSL) and retrieval-augmented generation (RAG). As discussed in Section 1.1, LLMs can analyze security and privacy violations by understanding the app's logic from the source code. However, due to input limitations, we cannot import the entire app's source code into LLMs. FSL and RAG help LLMs to better understand the app (for example, the app's intended use, the list of app functions, etc.) even if not all the code blocks of the app are provided as input.

1.2.1. LLMs

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 [Yigit et al., 2024].

GenAI [Zhang et al., 2023] is a subset of AI that excels in creating new content such as articles, poems, music, paintings, and films. LLMs [Minaee et al., 2024] is a subset of GenAI that primarily focuses on understanding human natural language and generating language-related material, including

translation, text summarization, and programming. LLMs can comprehend language's statistical and semantic characteristics using text-based training datasets.

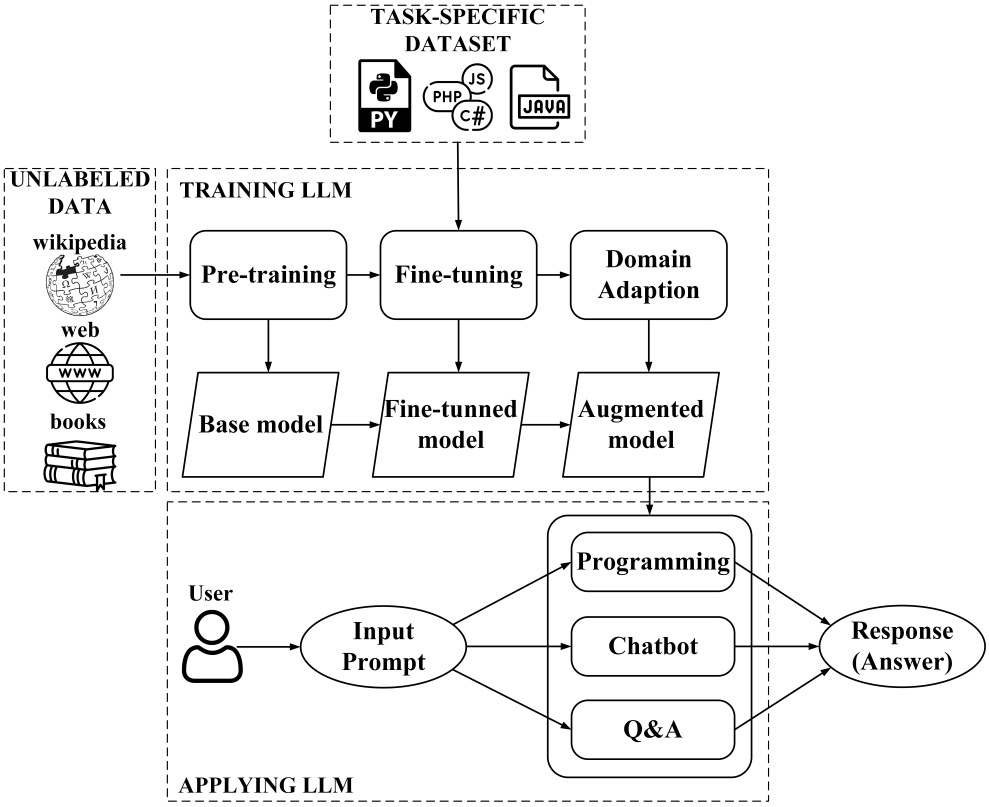

Figure 1.1: LLMs overview (Source: Tran Thanh Lam Nguyen)

Figure 1.1 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 [Naveed et al., 2023]. 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 [Gupta et al., 2023]. Pre-training uses unsupervised learning (without human intervention) to build a base LLM. For instance, OpenAI requires the base model (GPT) [Achiam et al., 2023] to predict the next word in an incomplete sentence, whereas Google trains BERT [Devlin et al., 2018] 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 [Roziere et al., 2023] is a fine-tuned model for programming-related tasks built on top of the base model Llama [Touvron et al., 2023]. 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[8] is designed explicitly for chatbots, based on variations of the GPT model.

To use LLMs, 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 LLMs' role to enhance the model's reasoning ability. For instance, we can formulate the prompt to instruct the LLMs 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*

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

language. Please help me implement a quick sort algorithm in Android".

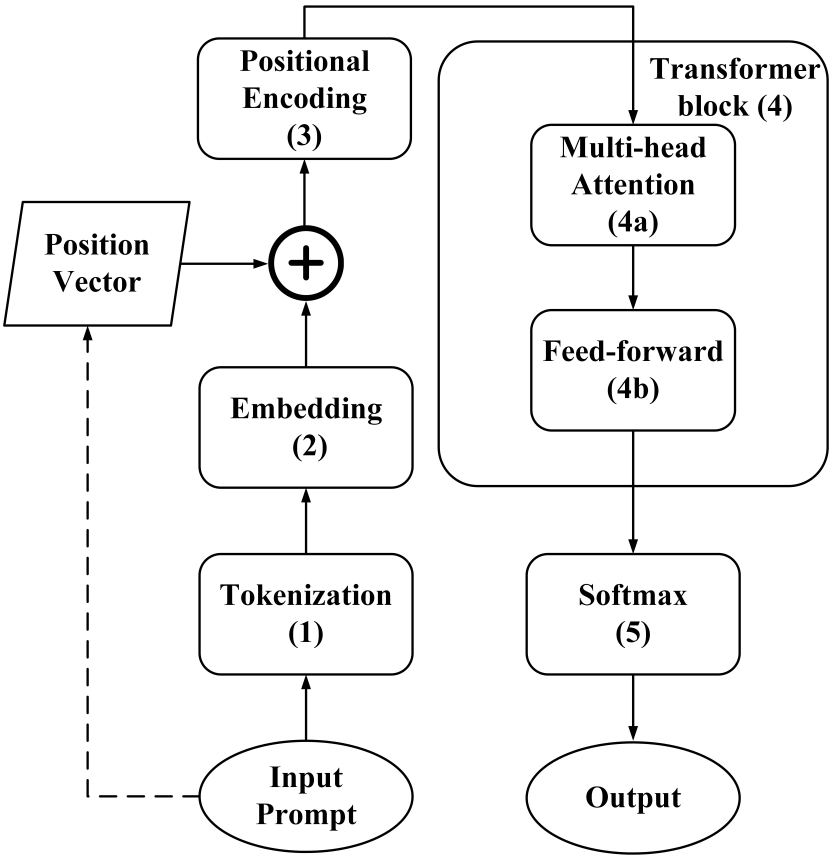

Figure 1.2: Transformer architecture [Vaswani, 2017]

It is easy to realize that training the base model is the foundation for effectively deploying LLMs, and it is the most expensive process. For instance, the training cost of GPT-3 is over 1 million USD, whereas for GPT-4 is over 100 million USD.[9]

Most of the current LLMs, such as OpenAI's GPT family (GPT-3, GPT-3.5, GPT-4) and Google's Gemini [Team et al., 2023], are based on the trans-

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

former architecture (see Figure 1.2) for training the base model [Pan et al., 2024], rather than employing older architectures like RNN (Recurrent Neural Network) and its variants [Sherstinsky, 2020]. Therefore, in this section, we focus 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 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 1.1 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, we choose the input prompt to be the incomplete sentence *"I go to school, and ..."* 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 1.2) 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 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 [Vaswani, 2017] 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 [Niu et al., 2021]. 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"]* → 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 architecture, 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"*.

In addition to answering questions, translating, and summarizing pure text, LLMs demonstrate excellent abilities in supporting coding activities. With this capability, LLMs can be an effective solution to replace traditional analysis methods in analyzing app security and privacy violations (cf. Section 1.4). More precisely, LLMs have many applications in the programming field, including (1) *Code search & document generation*: LLMs 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 1.1 summarizes existing code-specific LLMs along with their developer, supported programming languages, architecture, and main applications. Ac-

cording to Table 1.1, one can select the suitable LLMs 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, Codex has code summarization ability, which is more appropriate for understanding code logic and detecting security vulnerabilities.

Table 1.1: LLMs for code generation and analysis.

Model name & Developer	**Supported Programming Languages**	**Applications**
CodeBERT (Microsoft) [Feng et al., 2020]	Python, Java, JavaScript, PHP, Ruby, Go	Code search & document generation
GraphCodeBERT (Microsoft) [Guo et al., 2020]	Python, Java, JavaScript, PHP, Ruby, Go	Code search, Code clone detection, Code refinement, Code translation
InCoder (Meta AI) [Fried et al., 2022]	Python, JavaScript, Ruby, Go, Java, PHP	Code infilling
AlphaCode (DeepMind) [Li et al., 2022]	Python, C++	Code generation
GPT-3/GPT-3.5/ GPT-4 & variants (e.g., GPT-3.5-turbo, GPT-4 turbo,GPT-4o) (OpenAI) [Achiam et al., 2023]	Python, JavaScript, Go, Perl, PHP, Ruby, Swift, TypeScript, Shell	Code generation, Code summarization, Code refactoring, Code executing
Code Llama (Meta AI) [Roziere et al., 2023]	Python, C++, Java, PHP, Typescript, C#, Bash	Code generation, Code infilling

Although LLMs can understand natural language and programming languages, they still have certain limitations. For example, although Meta releases Llama 3.1 as open source, deploying this 405 billion-parameter model

requires at least 256GB of RAM, 1944GB of GPU, and 780GB of storage.[10] Of course, one can choose a model with fewer parameters, but this means accepting a trade-off in the model's prediction accuracy. Moreover, other models, e.g., GPT-3, GPT-4, and GPT-4o, are paid services, and service providers (OpenAI) charge fees for both the number of tokens input and output (i.e., LLM's responses).[11] Second, while training or fine-tuning LLMs for specific tasks is theoretically possible, creating a supervised dataset for specialized software engineering jobs is difficult and, thus, impractical for most individuals and firms [Ahmed et al., 2024]. Moreover, LLMs have limitation in the input prompts (for example, OpenAI's GPT-4 model[12] has a context window of only 128,000 tokens), thus, one cannot enter the entire source code into LLMs for code summarization or refactoring.

To bypass the token restriction, one must extract code blocks that are directly related to the targeted purpose. However, extracting only specific code blocks will reduce the context and thus reduce LLMs' performance. Finally, LLMs often give long answers and repeat parts of the question [Zhao et al., 2023], so there is a need for a mechanism to summarize their responses into formats such as JSON to easily evaluate the results. The above weaknesses are also experienced when LLMs are used to detect security/privacy risks in mobile apps.

[10] https://llamaimodel.com/requirements/
[11] https://openai.com/api/pricing/
[12] https://platform.openai.com/docs/models/gpt-4

1.2.2. FSL and RAG

In Section 1.2.1, we understand that the transformer architecture predicts output based on the input prompt. Therefore, the input prompt plays an important role in the performance of LLMs. In particular, for applying LLMs to identify security and privacy violations, the input prompt requires a lot of context related to the considered app. In addition, as discussed above, providing the entire source code of the app to the input prompt is not feasible due to token limitations. Therefore, in this section, we present FSL and RAG, two approaches to optimize the input prompt. Specifically, thanks to FSL and RAG, one can provide a few information to the LLM (e.g., code blocks related to the target vulnerabilities) to be able to analyze security and privacy risks. FSL and RAG are collectively known as **prompt engineering**.

Few-shot learning (FSL) [Song et al., 2023] 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) [Wang et al., 2020]. FSL learning is suitable for classification and recognition problems when collecting all training samples is impossible [Song et al., 2023].

More precisely, the FSL final prompt consists of two components, namely an example collection and a user's prompt. 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 LLMs, thus increasing the model's accuracy.

FSL is suitable for classification tasks, such as classifying scam emails. For

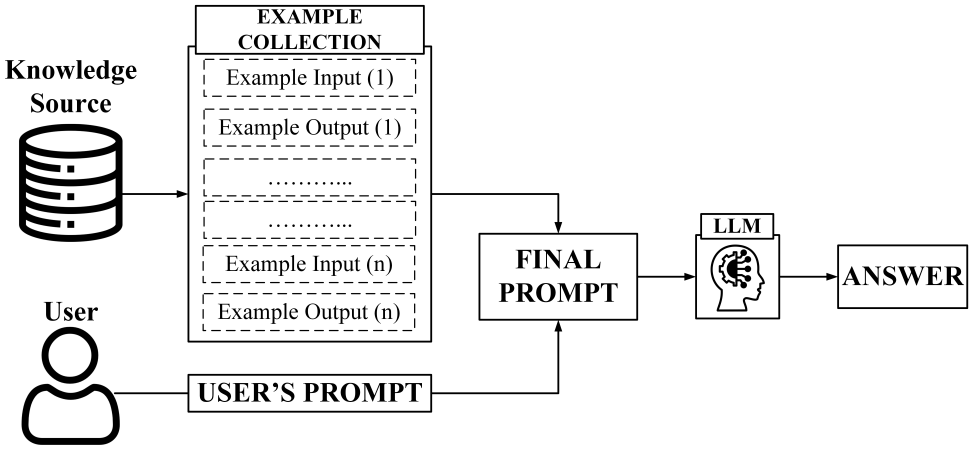

Figure 1.3: FSL overall architecture (Source: Tran Thanh Lam Nguyen)

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* $\{user's\ prompt\}$ *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 1.2.

However, FSL has two main disadvantages. First, we cannot provide too many examples in the FSL prompt since the limitation of LLMs token input, as discussed in Section 1.2.1. Second, the examples provided to the LLMs 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 experience of FSL users. This shortcoming may lead to other examples that are more relevant to the input prompt being overlooked.

Table 1.2: 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

Similar to FSL, RAG [Sawarkar et al., 2024] aims to optimize the input prompt for LLMs. However, RAG 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 1.4. In the preparation stage, labeled data (knowledge source) is transformed into embedding vectors via embedding models (e.g., OpenAI's text-embedding-ada-002[13]). 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[14] or ChromaDB[15]).

In the retrieval stage, suppose a user sends his/her prompt to LLMs. The

[13]https://platform.openai.com/docs/guides/embeddings/embedding-models
[14]https://github.com/facebookresearch/faiss
[15]https://www.trychroma.com/

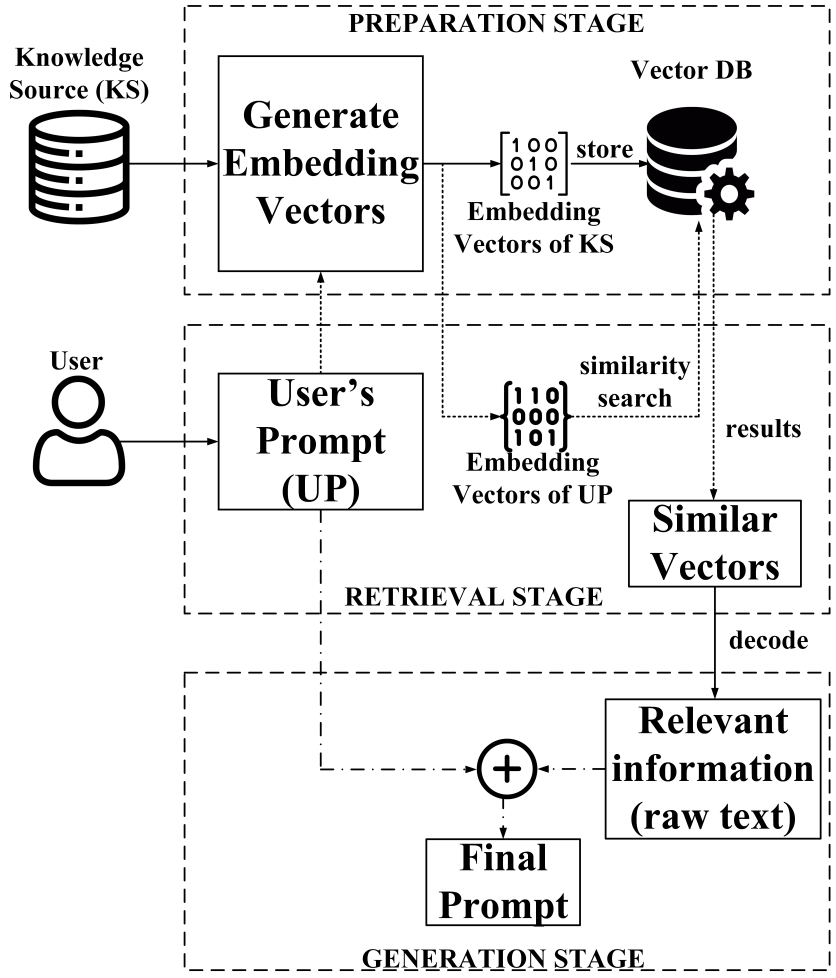

Figure 1.4: RAG overall architecture (Source: Tran Thanh Lam Nguyen)

user's prompt will first be converted into embedding vectors using the same embedding models used in the preparation stage. Next, the embedding vector of the user's prompt is used to query the vector DB 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 LLMs.

Next, the user's prompt (as raw text) is combined with relevant information to form the final prompt for the LLMs. 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 1.3 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*

Table 1.3: 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

contextual information $\{relevant\ information\}$". At this point, the LLMs 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.

In summary, both FSL and RAG help enhance the context of LLMs through optimized input prompt, thus balancing accuracy and token limitation. As a result, FSL and RAG can be applied in security and privacy analysis. Specifically, instead of providing the entire source code, we can provide labeled examples (FSL) or relevant information (RAG) along with a sufficient code segment in the input prompt for LLMs to analyze. RAG is more robust than FSL in enhancing context for LLMs, especially for complex tasks such as code summarization, as it is equipped with a mechanism to evaluate the relevance of

information for the input prompt. However, for simple tasks such as automatic interaction with the app's GUI or app classification (such as malware or benign apps), FSL is a suitable choice because it does not require the cost of maintaining the vector DB.

1.3. Mobile apps: main security and privacy threats

Mobile applications 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, licensing models, and more, to carry out user attacks.

In this section, we describe the top 10 security and privacy issues associated with mobile apps according to the OWASP Top 10 2024 report.[16] OWASP (Open Web Application Security Project) is an international non-profit organization specializing in web application security that, in recent years, has extended its scope to mobile apps to raise awareness of their security/privacy vulnerabilities.

Then, in Section 1.4 we discuss some research proposals applying LLMs to mitigate some of the risks mentioned in the OWASP Top 10 2024.

R1: Improper Credential Usage (Exploitability level: Easy): Some

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

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 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[17] 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

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

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, 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[18], DES[19], Triple DES[20], or SHA-1[21], 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 1.4 reports for each of the above explained risks an exemplary paper showing the practical impact of the risk.

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

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

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

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

Table 1.4: Security risks and examples of impacts.

Risk	Examples of impact
R1	[Zhou et al., 2015] shows that 51.5% (121/237) of the analyzed apps leaked email service keys, & 67.3% (132/196) leaked Amazon AWS API keys because the developer hardcoded authentication information.
R2	[Wen et al., 2018] shows that 66% of 100 popular iOS apps leaked WeChat SDK credentials & 37% for Weibo SDK (two popular SDKs with over 40 million users).
R3	[Wang et al., 2015] shows that 86.2% of 4000 apps in the Chinese market have deviated implementation of OAuth 2.0 from the standard recommended by RFC
R4	[Bao et al., 2017] shows that hybrid Android apps (developed by using web programming languages but can run on Android OS) are vulnerable to XSS attacks .
R5	[Gadient et al., 2021] shows that - 47.8% of URLs in 303 open-source apps & 69.3% in 3,073 closed-source apps use HTTP (unencrypted). - 67.7% of URLs in open-source apps & 88.3% in closed-source apps lack HSTS implementation (force redirect from HTTP to HTTPS).
R6	[Nguyen et al., 2022] 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), & 1% shared data despite user refusal).
R7	[Huang and Chen, 2022] found that image classification apps store AI models (i.e., parameters for deep learning models) directly in the app source code. This leads to the risk of adversarial attacks. The author successfully attacked 71.7% of AI models in 114 image classification apps.
R8	[Li et al., 2020] 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	[Reardon et al., 2019] shows that Baidu and Salmonads SDK exploited the SD card as a covert channel to access the phone's IMEI (International Mobile Equipment Identity).
R10	[Yoshida et al., 2018] analyzed over a million apps, identifying 223 apps signed with a weak 512-bit RSA key & 52,866 apps using the insecure MD5 algorithm.

1.4. LLM-based solutions: state of the art

Section 1.3 shows that attack scenarios on mobile apps are becoming more frequent and sophisticated. Traditional analysis methods such as static, dynamic, and hybrid analysis have many weaknesses that gradually make them insufficient to cover increasingly complicated attacks.

In this section, we present state-of-the-art research on applying LLMs to the security and privacy aspects of mobile apps. Since several of the surveyed research proposals can be applied to analyze one or more mobile app 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, bug detection/reproduction*, and *malware detection*. Although the application goals differ, all these research proposals leverage LLMs to analyze the app source code. One of their main contributions is how to optimize the input prompt for LLMs to target the addressed scenario. Indeed, as discussed in Section 1.2, retraining LLMs is only feasible in theory, so the input prompt plays a decisive role in the performance of LLMs. 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 section, we will provide a table (cf. Table 1.6) mapping the presented state-of-the-art solutions to the addressed OWASP risks (presented in Section 1.3).

Please note that when presenting the performance of the various approaches, the reported results are taken from the corresponding research paper and are only for reference purposes, aiming to provide an initial view of the feasibility

of using LLMs for the described target scenario.

1.4.1. Vulnerabilities detection

As discussed in Section 1.3, mobile apps contain many security/privacy vulnerabilities that can lead to various attack scenarios that are hard to cover with traditional analysis methods. LLM, with its excellent code summarization capabilities, can be used to automatically identify some of those security/privacy vulnerabilities.

One paper in this area is the one by Kouliaridis et al. [Kouliaridis et al., 2024], which evaluated the performance of nine LLMs (opensource and paid), namely GPT-3.5, GPT-4, GPT-4 Turbo,[22] Llama-2,[23] Zephyr Alpha,[24] Zephyr Beta,[25] Nous Hermes Mixtral,[26] Mistral Orca,[27] and Code Llama,[28] in identifying the OWASP Mobile Top 10 vulnerabilities presented in Section 1.3. In addition, the authors compared the code summarization capabilities of these nine LLMs with two commonly used code analysis tools, namely Bearer[29] and MobSF.[30] 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)

[22]https://platform.openai.com/docs/models/gpt
[23]https://www.llama.com/llama2/
[24]https://huggingface.co/HuggingFaceH4/zephyr-7b-alpha
[25]https://huggingface.co/HuggingFaceH4/zephyr-7b-beta
[26]https://ollama.com/library/nous-hermes2-mixtral
[27]https://ollama.com/library/mistral-openorca
[28]https://ollama.com/library/codellama
[29]https://github.com/Bearer/bearer
[30]https://github.com/MobSF/Mobile-Security-Framework-MobSF

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 can not 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 LLMs perform better than the current popular static analysis tools.

Table 1.5 lists the LLMs with the best detection performance (the highest D score) for each specific vulnerability.

Table 1.5: Performance of LLM models for specific vulnerabilities [Kouliaridis et al., 2024].

OWASP Risk	**Best LLMs**
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

Gabriel Morales et al. [Morales et al., 2024] 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 - extract-

ing 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 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[31] 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

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

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. [Oishwee et al., 2024] examined the potential of LLMs to help developers address the complexity of Android permissions (cf. R8 in Section 1.3). Specifically, the paper compared ChatGPT (model GPT 3.5) answers with accepted answers on Stack Overflow[32] (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

[32]https://stackoverflow.com/

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.

1.4.2. 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 1.1. Monkey [33] is a commonly

[33]https://developer.android.com/studio/test/other-testing-tools/monkey

used tool for automated interaction with app's GUIs in many research papers (e.g., [Reardon et al., 2019, Yerima et al., 2019]). However, this tool generates only random inputs rather than context-specific test scenarios. It cannot bypass apps requiring registration or login to activate deeper functionalities, leading to a high rate of false negatives [Andow et al., 2020, Han et al., 2020]. 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. [Liu et al., 2024a] designed GPTDroid, a tool based on ChatGPT model GPT-3.5-turbo for GUI automated testing. The primary objective of GPTDroid is to 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 LLMs to generate and execute test scenarios. At the same time, the apps' responses are sent back to LLMs 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 LLMs can be a valuable alternative to existing tools (such as Monkey) to improve the scalability of dynamic analysis.

Zhe Liu et al. [Liu et al., 2024b] 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 1.3).

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 1.2.2) 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.

1.4.3. 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 the favor-

able 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. [Zhao et al., 2025] 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 1.2).

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.[34] 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

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

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, LLMs 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 1.6 reports for each of the papers described above the used LLMs, the main research targets, and the OWASP risks it addresses.

The state-of-the-art research presented above is a valuable starting point for the adoption of LLMs to mitigate security and privacy risks in mobile apps. The research results are promising, contributing to opening new paths for applying LLMs to investigate more attack scenarios in the future.

Table 1.6: Surveyed LLM-based approaches and related OWASP risks.

Category	**Paper**	**Used LLMs**	**Research target**	**OWASP Risks**
Vulnerabilities detection	[Kouliaridis et al., 2024]	GPT-4 & Code Llama	Identify vulnerabilities in the app & propose remediation solutions	R1 to R10
	[Morales et al., 2024]	ChatGPT (GPT-3.5)	Examine the actual app's behavior & its privacy policies	R3,R5,R6, R8 & R9
	[Oishwee et al., 2024]	ChatGPT (GPT-3.5)	LLMs assist developers in managing complex Android permissions	R8
Bug Detection & Reproduction	[Liu et al., 2024a]	ChatGPT (GPT-3.5-turbo)	App GUI automated testing.	R2,R3,R4, R5,R6 & R9
	[Liu et al., 2024b]	ChatGPT	Generates invalid text inputs to detect bugs.	R4
Malware Detection	[Zhao et al., 2025]	GPT-4	Malware detection	R1,R2,R3,R5, R6,R8 & R9

1.5. An LLMs-based Approach for Mitigating Image Metadata Leakage Risks

In Section 1.4, we discussed how state-of-the-art proposals apply LLMs to address some of the security and privacy risks in mobile apps. In this section, as an illustrative example, we show how LLMs can be applied to identify security and privacy infringements in a particular attack scenario: a side-channel attack that results in the exposure of sensitive information when users share images online (e.g., upload the image to cloud storage, send the image via instant message apps, or share images on a social network) through images metadata. This is a dangerous risk because taking and sharing images is a common habit of smartphone users [Lam et al., 2024].

When taking a picture, it is a standard for digital cameras (including smartphones) to automatically generate metadata information about the date and time the image was taken, camera specification, GPS location, camera focal length parameters, etc. This information is usually encoded using EXIF (Exchangeable Image File Format).[35] EXIF metadata is embedded automatically into the image and does not display visually, so users rarely notice it exists. However, malicious users can compromise users' security and privacy by exploiting sensitive information contained in EXIF metadata, such as the date and time of capture, camera model, camera brand, device serial number, and GPS coordinates. There are many studies demonstrating that malicious users

[35]https://www.media.mit.edu/pia/Research/deepview/exif.html

can take advantage of those sensitive metadata to execute different kind of attacks, for example, social engineering attacks [Tayeb et al., 2018, Faircloth, 2017], spoofing attacks [Gouert and Tsoutsos, 2022], and re-identification attacks [Pratik and Sendhil, 2023].

In [Lam et al., 2024], we show how EXIF metadata can be used to expose sensitive user information when users share images. Indeed, EXIF metadata contains sensitive information hidden in the image, so users are unaware of its existence and do not remove sensitive metadata before sharing the image online. Additionally, EXIF metadata is not protected by the Android's permission model (cf. the R6 risk we mentioned in Section 1.3). Finally, images stored in public storage are not protected by the sandbox mechanism (cf. the R9 security vulnerability we presented in Section 1.3). Specifically, Android requires applications to obtain user consent through a permission mechanism before accessing personal information or using the device's hardware. For example, an app must be granted *ACCESS_FINE_LOCATION* permission when accessing location and *BLUETOOTH_CONNECT* for Bluetooth connection. However, Android does not provide permission to protect EXIF metadata. In addition, images (after being captured) are stored in public storage instead of isolated by the sandbox mechanism (cf. risk R9 in Section 1.3). Therefore, an app can access images (generated by other apps) and read EXIF sensitive metadata embedded inside the image without the user's explicit consent if they are granted read/write storage permission.

Figure 1.5 illustrates the considered threat model. First, a user uses App-A (e.g., a camera app) to take photos. These photos contain EXIF metadata and are stored in public storage. Because mobile devices usually have limited storage capacity, the user uses App-B (e.g., Microsoft One Drive) to upload

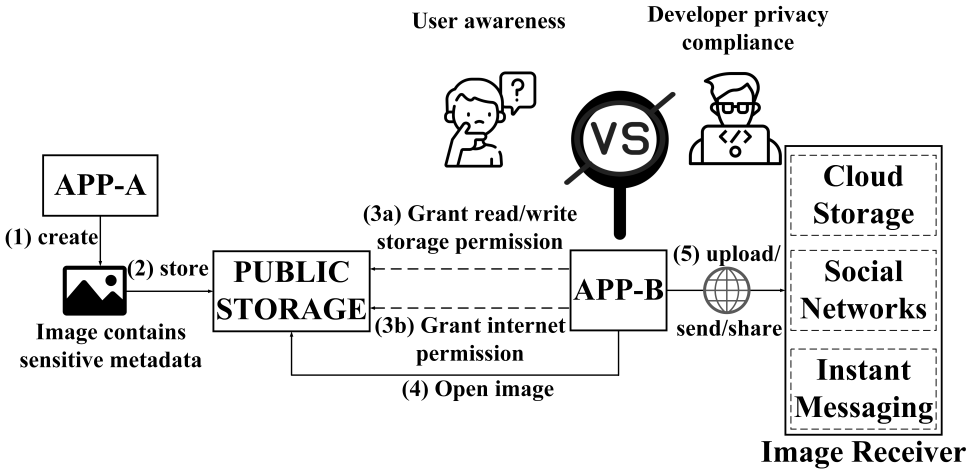

Figure 1.5: Threat model for sensitive data leakage through EXIF metadata [Lam et al., 2024]

images to cloud storage for long-term storage. To upload images, the user must grant read/write storage and internet permissions to App-B. If App-B's sent-out traffic contains sensitive metadata, meaning App-B does not proactively remove sensitive metadata before sending the information out of the phone, the user may face security/privacy risks due to the leakage of sensitive information from the image's metadata. In this threat model, public storage is considered a side channel for the unauthorized collection of sensitive information. Indeed, our observations, reported in [Lam et al., 2024], show that One Drive sent five types (namely, datetime, smartphone model, smartphone brand, device serial number, and GPS) of sensitive metadata instead of removing them.

To verify the frequency of the above mentioned threat, we developed MetaLeak [Lam et al., 2024], a framework based on hybrid analysis. We then downloaded 43,718 apps from APK combo[36] belonging to various categories,

[36]https://apkcombo.app/

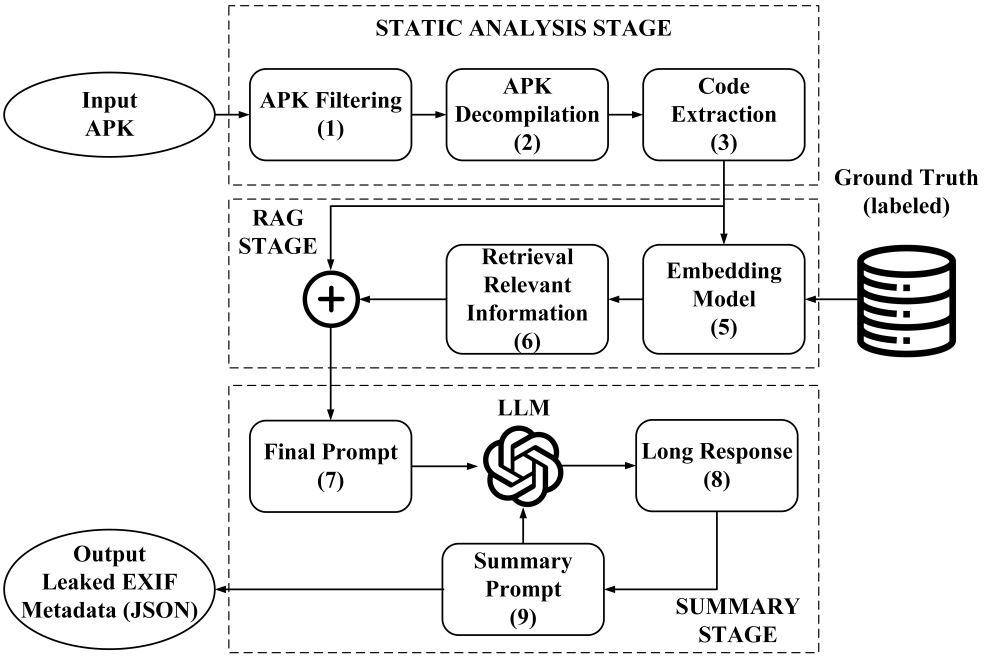

Figure 1.6: LLMs-based proposal workflow (Source: Tran Thanh Lam Nguyen)

such as photography, communication, productivity, etc. Through Metaleak, we performed static analysis over them to collect the Manifest.xml of the apps. Next, we filtered and retained only those apps that required 3 permissions, namely read storage (READ_EXTERNAL_STORAGE), write storage (WRITE_EXTERNAL_STORAGE), and internet access (INTERNET), because these permissions are necessary to access images in public storage and share images online. After the static analysis, we obtained 26,230 apps. Next, we selected 5000 popular apps based on the number of installations and performed dynamic analysis over them. We installed the apps on a rooted AVD (Android Virtual Device) and used an MITM (man-in-the-middle) proxy to observe the app's sent-out traffic when users shared images online. The results showed that 21.9% (1095/5000) of the considered apps leaked at least one type of sensitive metadata (1071/1095 apps leaked four or five types of sensi-

tive metadata, while only 24/1095 apps leaked just one type). Specifically, 680 apps leaked GPS information, 1043 apps leaked datetime, 1055 apps leaked smartphone model, 1055 apps leaked smartphone brand, and 998 apps leaked device serial number.

However, MetaLeak has a significant weakness, which is the semi-automation workflow, specifically during dynamic analysis. Although we tested many automation tools to interact with the app's GUI, none of them fit the MetaLeak use case. Therefore, app evaluation has been done manually by recruited testers, who spent an average of 180 seconds per app. Testers having to repeat boring steps in dynamic analysis is a big scalability issue.

Thus, in what follows, we discuss an architecture that leverages LLMs to eliminate the need for app dynamic analysis. Figure 1.6 illustrates the LLMs-based solution we envision, consisting of three main steps, namely (1) static analysis, (2) RAG, and (3) summary. In the static analysis stage, apps' APKs are processed in two steps. In step 1, APK filtering (1), we extract the app's manifest file and only keep those apps that require read/write storage and internet permissions because these permissions are necessary to access and share images online following the threat model. Next, we verify the app's support for image uploading by examining the mineType parameter in the AndroidManifest.xml file to reduce the number of apps that require analysis. In fact, apps that require read/write storage and internet permissions but do not support image sharing (e.g., only support sharing PDF files) are irrelevant to the considered threat model. In step 2, APK decompilation (2), we decompile APK files that satisfy the previously mentioned permission and mimeType settings to acquire the app's source code. Following this, the code extraction module (3) identifies keywords, such as the name of the EXIF metadata han-

dling method (e.g., getDatetime, getGPS), in the app's source code to extract EXIF-related code blocks. The purpose of extracting EXIF-related code blocks is because of the input token limit of LLMs. In the RAG stage, we follow the RAG workflow as described in Section 1.2.2. Specifically, the dataset we collected in [Lam et al., 2024] is used as ground truth. The ground truth data is transformed into an embedding vector using an embedding model (5) (e.g., OpenAI's text-embedding-3-large[37]) and then stored in the vector database. Subsequently, the EXIF-related code blocks of the app are also transformed into embedding vectors using the same embedding model as the ground truth data to retrieve relevant information from the vector database (6). Finally, the retrieved information is combined with the EXIF-related code blocks to form the final prompt (7). The final prompt is sent to the LLM to determine how the code block handles EXIF metadata, exploiting its the code summarization capability. Specifically, the LLMs will decide if the code block will remove or retain sensitive metadata and what metadata type will be retained.

However, because LLMs' responses are often lengthy and repeat part of the question (i.e., long response (8)) [Zhao et al., 2023], the architecture contains a summary stage, where a summary prompt (9) is created The summary prompt is then sent to LLM to generate the final JSON response.

1.6. Research Challenges

LLMs have great potential to address mobile app security risks and privacy violations. However, there is no generalized plug-and-play solution for LLMs.

[37]https://openai.com/index/new-embedding-models-and-api-updates/

In particular, all state-of-the-art research proposals we discussed in this paper aim to optimize LLM input prompts, which require expert knowledge in security and privacy. 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. In addition, the usage and operating costs remain high. 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, but there is no guarantee that this code block is executed at run-time. 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 1.1) but is not yet widely adopted.

Other relevant open research issues, when applying LLMs to mitigate security and privacy risks of mobile apps are discussed in what follows.

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. [Lin et al., 2024] found that unverified LLMs can be packaged into malicious services. Currently, there are many LLMs as malicious services introduced on the black market, mainly providing features such as malicious code generation, phishing emails, and scam sites, as summarized in Table 1.7. 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 themselves steal information from the people who use them

Table 1.7: Malicious LLMs [Lin et al., 2024].

Name	Malicious code	Phishing Emails	Scam Site
CodeGPT	✓	×	✓
MakerGPT	✓	×	✓
FraudGPT	✓	✓	✓
WormGPT	✓	✓	✓
XXXGPT	✓	×	×
WolfGPT	✓	✓	✓
Evil-GPT	✓	✓	✓
DarkBERT	✓	✓	×
DarkBARD	✓	✓	×
BadGPT	✓	✓	✓
BLACKHATGPT	✓	×	×
EscapeGPT	✓	✓	✓
FreedomGPT	✓	✓	✓
DarkGPT	✓	✓	✓

- 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 1.7 describes content poisoning attacks on 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 risk 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 answer of the manipulated LLMs, their API key will be stolen.

Finally, the permission mechanism of mobile apps (both Android and iOS) has a weakness in that it grants normal users—who often lack knowledge of security and privacy—the authority to grant dangerous permissions to apps for accessing sensitive information. Indeed, users have to make decisions without any assistance [Lam et al., 2024]. Therefore, applying LLMs to build chatbots embedded directly on mobile OS can support users in making decisions about granting permissions to mobile apps. For example, chatbots can support summarizing the app's data safety[38] information (i.e., the type of data the app will collect and share with 3rd party) and notify users when they install it. Moreover, LLMs can perform analysis of the app's source code directly on the phone (i.e., static analysis and code summarization) or monitor app behavior (e.g., the type of sensitive data the app frequently shares) to warn users about potential risks [Chen et al., 2024].

1.7. Conclusion

In this chapter, we present the potential of LLMs in analyzing security and privacy violations in mobile apps. With their great potential, LLMs can complement traditional methods such as static analysis, dynamic analysis, or hybrid analysis, which have many weaknesses and make it challenging to cover all the current sophisticated vulnerabilities. In the chapter, we discussed how to apply LLMs to limit security and privacy risks, including vulnerabilities detection,

[38]https://developer.android.com/google/play/integrity/other

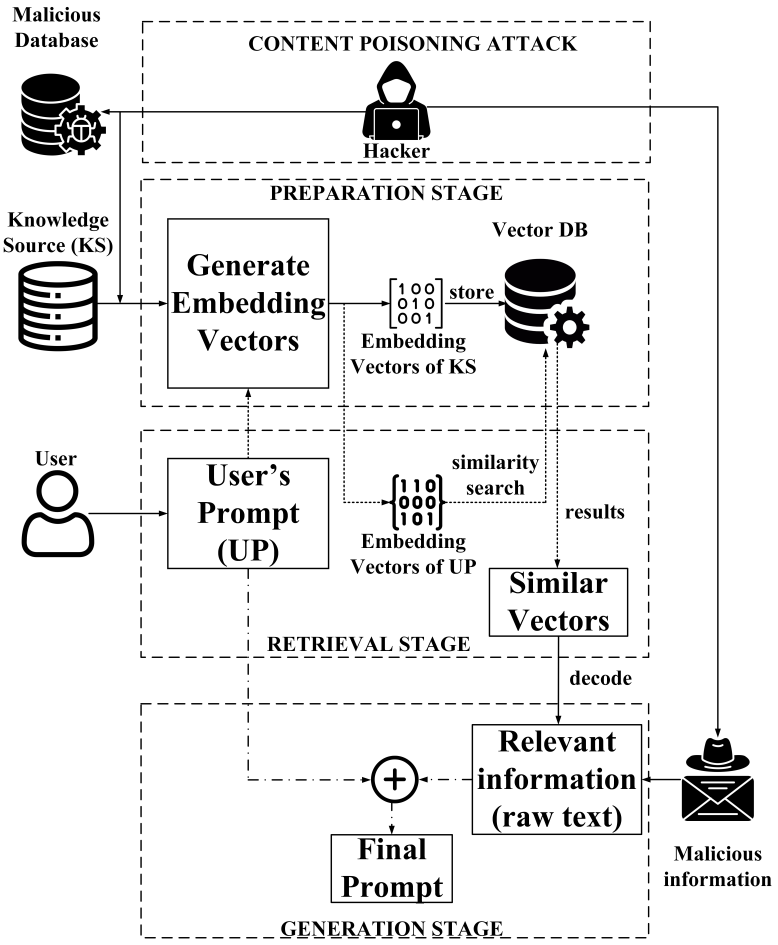

Figure 1.7: Adversarial Attacks to RAG (Source: Tran Thanh Lam Nguyen)

bug detection & reproduction, and malware detection through the analysis of state-of-the-art research. However, pure LLM-based approaches are not a solution for all issues because they still have several weaknesses. Therefore, regardless of the application purpose, optimizing the input prompt to provide sufficient information and context for LLMs reasoning is one of the key goal. In the chapter, besides describing an illustrative example of the usage of LLMs to target a security weakness in Android that results in the leakage of sensi-

tive information when users share images online, we also discuss open research issues in the field.

Acknowledgment

The work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

Bibliography

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.

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.

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.

Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. Actions speak louder than words:{Entity-Sensitive} privacy policy and data flow analysis with

{PoliCheck}. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 985–1002, 2020.

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.

Daihang Chen, Yonghui Liu, Mingyi Zhou, Yanjie Zhao, Haoyu Wang, Shuai Wang, Xiao Chen, Tegawendé F Bissyandé, Jacques Klein, and Li Li. Llm for mobile: An initial roadmap. *ACM Transactions on Software Engineering and Methodology*, 2024.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina N. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018. URL https://arxiv.org/abs/1810.04805.

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. ISBN 978-0-12-802149-1. doi: https://doi.org/10.1016/B978-0-12-802149-1.00008-7. URL https://www.sciencedirect.com/science/article/pii/B9780128021491000087.

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

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.

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.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

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.

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

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

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.

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.

Catherine Han, Irwin Reyes, Álvaro Feal, Joel Reardon, Primal Wijesekera, Narseo Vallina-Rodriguez, Amit Elazari, Kenneth A Bamberger, and Serge Egelman. The price is (not) right: Comparing privacy in free and paid apps. *Proceedings on Privacy Enhancing Technologies*, 2020.

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.

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.

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

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

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.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097, 2022.

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.

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, 2024a.

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, 2024b.

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.

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.

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.

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.

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

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.

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. doi: 10.1109/TKDE.2024.3352100.

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.

Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An ex-

ploration of apps' circumvention of the android permissions system. In *28th USENIX security symposium (USENIX security 19)*, pages 603–620, 2019.

Deepjyoti Roy and Mala Dutta. A systematic review and research perspective on recommender systems. *Journal of Big Data*, 9(1):59, 2022.

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.

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.

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.

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

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.

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.

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.

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.

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.

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.

A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

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.

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.

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.

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.

Suleiman Y Yerima, Mohammed K Alzaylaee, and Sakir Sezer. Machine learning-based dynamic analysis of android apps with improved code coverage. *EURASIP Journal on Information Security*, 2019(1):1–24, 2019.

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

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. doi: 10.1109/COMPSAC.2018.10324.

Chaoning Zhang, Chenshuang Zhang, Sheng Zheng, Yu Qiao, Chenghao Li, Mengchun Zhang, Sumit Kumar Dam, Chu Myaet Thwal, Ye Lin Tun, Le Lu-

ang 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.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

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.

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.