What's Done Is Not What's Claimed: Detecting and Interpreting Inconsistencies in App Behaviors

Chang Yue[1,2], Kai Chen[1,2,⋆], Zhixiu Guo[1,2], Jun Dai[3], Xiaoyan Sun[3] and Yi Yang[1,2]

[1]Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Department of Computer Science, Worcester Polytechnic Institute, USA
{yuechang,chenkai,yangyi}@iie.ac.cn, gzhixiu@gmail.com, {jdai,xsun7}@wpi.edu

Abstract*—The widespread use of mobile apps meets user needs but also raises security concerns. Current security analysis methods often fall short in addressing user concerns as they do not parse app behavior from the user's standpoint, leading to users not fully understanding the risks within the apps and unknowingly exposing themselves to privacy breaches. On one hand, their analysis and results are usually presented at the code level, which may not be comprehensible to users. On the other hand, they neglect to account for the users' perceptions of the app behavior. In this paper, we aim to extract user-related behaviors from apps and explain them to users in a comprehensible natural language form, enabling users to perceive the gap between their expectations and the app's actual behavior, and assess the risks within the inconsistencies independently. Through experiments, our tool *InconPreter* is shown to effectively extract inconsistent behaviors from apps and provide accurate and reasonable explanations. InconPreter achieves an inconsistency identification precision of 94.89% on our labeled dataset, and a risk analysis accuracy of 94.56% on widely used Android malware datasets. When applied to real-world (wild) apps, InconPreter identifies 1,664 risky inconsistent behaviors from 413 apps out of 10,878 apps crawled from Google Play, including the leakage of location, SMS, and contact information, as well as unauthorized audio recording, etc., potentially affecting millions of users. Moreover, InconPreter can detect some behaviors that are not identified by previous tools, such as unauthorized location disclosure in various scenarios (e.g. taking photos, chatting, and enabling mobile hotspots, etc.). We conduct a thorough analysis of the discovered behaviors to deepen the understanding of inconsistent behaviors, thereby helping users better manage their privacy and providing insights for privacy design in further app development.

I. Introduction

The global population of smartphone users has exceeded 5 billion [1]. This extensive usage has led to significant amounts of private data stored in users' mobile devices, including contacts, photos, and locations. However, concerns about app privacy are growing, as evidenced by Facebook's secret collection of call records and SMS texts without user consent [2]. In response, industries have enacted regulations such as the General Data Protection Regulation (GDPR), which introduces principles like "only processing necessary data" and "transparency in data processing" [3]. App platforms have implemented measures such as permission management systems to protect user data. However, these measures are not always sufficient [4–7]. Users often struggle to understand the permissions they grant, and apps may misuse data in ways that contradict user expectations. For example, users might permit a photo editing app to access location data but the app secretly uploads their location without clear notification. This inconsistency between app behavior and user expectations raises significant concerns regarding privacy and trust.

Prior work has made efforts to identify such inconsistencies [8–15]. However, these efforts have certain limitations: (1) *Incapable of fulfilling user's expectations.* Some studies rely solely on app function descriptions or analyze specific texts or icons on the UI to represent the functionalities disclosed to users. However, such information may not fully represent every user's expectations because different users may have varying understandings of the UI and app functionalities; (2) *Reliance on inaccurate assumptions.* Some works utilize neural network models to classify inconsistent behaviors, and they assume that the functionalities depicted by UI elements align with their actual functions in benign apps when obtaining training data. Nevertheless, this assumption is demonstrated to be flawed in previous work [15]; (3) *Focus on limited behaviors.* Some studies only identify inconsistencies related to behaviors associated with interactive UI elements such as buttons, overlooking numerous app behaviors that can be triggered without user interactions.

Approach. In our work, we do not rely on the inaccurate assumption that UI functionality description matches their actual functions, or restrict the analysis to behaviors associated with only interactive UI elements. Instead, we aim to extract user-related behaviors directly from apps and explain them to users in comprehensible natural language. This allows users to independently identify inconsistencies between their expectations and the apps' actual behaviors, assess the associated risks, and determine whether these risks are a concern for them. We develop *InconPreter* to achieve these objectives.

InconPreter first extracts a wide range of behaviors from the

⋆ Corresponding Author

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.242290
www.ndss-symposium.org

app. A *behavior* is defined as a call sequence accompanied by its associated UI context. Common methods that extract call graphs directly based on control flow graphs may not accurately represent a complete behavior due to missing implicit calling relationships and overlooking logical relationships during data processing. Therefore, InconPreter collects a list of common implicit calls and utilizes data flow analysis to further extend the call graph, constructing more comprehensive behaviors. InconPreter further filters and retains the behaviors that might concern users from all the extracted behaviors, based on observations of user data operation-related API naming conventions from various perspectives. By comparing the semantics of the code with the corresponding UI information, InconPreter identifies the inconsistent behaviors.

To help users comprehend these behaviors, identify the associated risks, and assess their concerns independently, InconPreter interprets the extracted inconsistent behaviors into understandable natural language. Given the excellent performance of large language models (LLMs) in fields like code understanding [16, 17], we leverage LLMs to interpret these behaviors. However, due to the redundant information in long call sequences, directly applying LLMs to these sequences presents significant challenges. In Android, API permissions effectively reflect the operations of APIs on system resources [18–20], and the number of permissions involved in an API sequence is relatively small, which makes it easier for LLMs to understand the behavior. Therefore, by designing appropriate prompts, InconPreter leverages LLMs to generate natural language interpretations of these behaviors based on permissions involved in the API sequence. Additionally, recognizing users' limited knowledge of security and privacy, InconPreter provides risk analysis to users for reference.

Results and Findings. InconPreter is shown to effectively extract inconsistent behaviors from apps and provide accurate and reasonable explanations. InconPreter achieves an inconsistency identification precision of 94.89% on our labeled dataset, and a risk analysis accuracy of 94.56% on the widely used Android malware datasets. User studies confirm that InconPreter's natural language interpretations for inconsistent behaviors are both reasonable and easily comprehensible, aiding ordinary users in understanding app behaviors.

InconPreter discovers 1,664 risky inconsistent behaviors from 413 apps out of 10,878 apps (after deduplication) crawled from Google Play between 2020 and 2024. These 413 apps are widely distributed and have over 4.3 billion downloads. The risky behaviors include leakage of location, SMS, and contact information, as well as unauthorized audio recording, etc., potentially affecting millions of users. Moreover, InconPreter detects some behaviors overlooked by previous tools, such as unauthorized location disclosure in various scenarios like taking photos, chatting, and enabling mobile hotspots. We analyze the distribution of risky inconsistent behaviors from different perspectives. For example, "Tools" category apps have the highest number of risky inconsistent behaviors, and accessing device location is one of the most common risky behaviors. Additionally, we find that 77.97% of the 413 apps contain self-starting risky inconsistent behaviors, indicating that without user interaction, apps automatically conduct a series of risky behaviors, posing serious threats to users and device privacy and security. Additionally, we explore the distribution of risky inconsistent behaviors in apps from 2010 to recent years. The results indicate a shifting landscape in the popularity rankings of risky inconsistent behaviors over time. Notably, the proportion of apps containing risky inconsistent behaviors decreased from 26.44% around 2010 to 3.80% in recent years. The trend underscores a positive trajectory toward enhanced mobile app privacy and security measures.

Contributions. The contributions of the paper are as follows:

• We analyze inconsistencies between user expectations and app behavior from users' perspective. Without inferring user expectations from UI elements, we explain sensitive app behaviors to users in understandable natural language, which bridges the gap between users' actual expectations and app behavior, enabling users to independently assess inconsistencies.

• We develop InconPreter to automatically extract user-related behaviors from apps and interpret these behaviors. To accurately extract behaviors that need attention from users, we design multiple behavior filters based on a series of insights and statistics. Then we carefully design prompts for the LLM to make the behavior explanations more readable and reasonable.

• From 10,878 wild apps crawled from Google Play, we discover 1,664 risky inconsistent behaviors from 413 apps, including leakage of location, SMS, and contact information, as well as unauthorized audio recording, etc., potentially affecting millions of users. We explore the distribution of these behaviors, which deepens the understanding of them, helping users better manage their privacy and providing insights for privacy design in further app development.

II. Background and Related Works

A. Android Security Analysis

Android analysis techniques can be primarily categorized into static analysis [21–27], dynamic analysis [28–33], and learning-based analysis [34–38]. However, most existing tools can only detect generic patterns of malicious behavior, such as the use of private data, without fully confirming whether such behavior indeed poses a risk. For example, Flowdroid [21] utilizes data flow analysis to trace paths between sources and sinks to identify potential sensitive data leaks. However, leakage of sensitive data does not inherently indicate malicious intent, as many benign apps also require access to sensitive information. IntelliDroid [33] design inputs for dynamic analysis, dynamically triggering and validating potential malicious behaviors present in the app. However, it does not consider whether such malicious behavior is unintentional from the user's perspective or if it indeed has an impact on users.

Some works identify risky behaviors from policy/permission's perspective [19, 39, 40]. Felt et al. [19] study Android applications to evaluate whether Android developers follow the principle of least privilege in their permission requests. Slavin et al. [40] propose a semi-automated framework that consists of a policy terminology-API method map that links policy

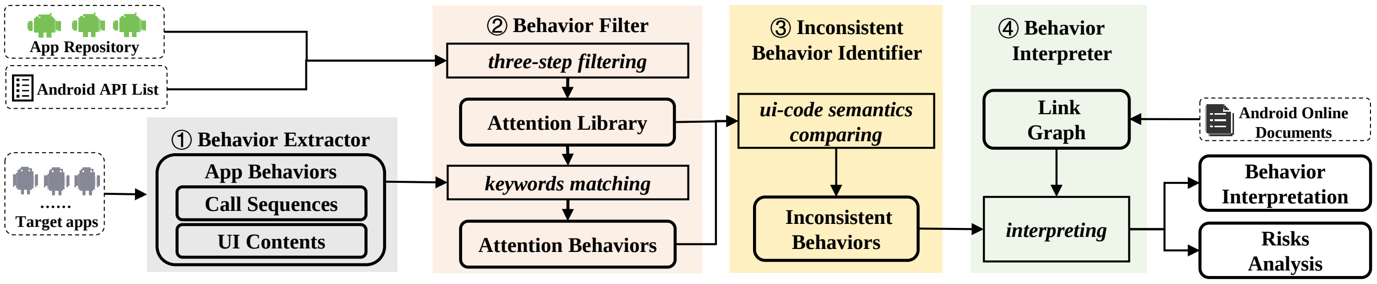


Fig. 1: The framework of InconPreter

phrases to API methods that produce sensitive information, and information flow analysis to detect misalignments. Zimmeck et al. [39] combine machine learning (ML) and static analysis techniques to analyze apps' potential non-compliance with privacy requirements. However, determining whether a behavior is truly risky to users is challenging. An app may comply with policies/permissions but still perform sensitive actions without user awareness. For example, users may consent to a social app's policy that allows the collection of their location for sharing moments, unaware that the app continuously tracks their location in the background, even when they are simply browsing stories or chatting with friends.

To address this issue, some researchers consider analyzing the risks within apps from the user's perspective, focusing on whether the functionality performed by the app violates the user's expectations [8–15]. Researchers [8, 9] use app descriptions as user expectations, but the granularity of the functional description is too coarse to express the different expectations of users in different scenarios. Several studies[10, 12, 13] use UI elements to infer user expectations. UI is more representative of user expectations than app description. However, they use the UI information in benign applications with the actual behavior performed by the apps as norm standards to detect abnormal behavior. As mentioned above, behaviors in benign applications do not always exactly match user expectations either, which has been demonstrated in related works [15]. In addition, they can only discover inconsistencies related to behaviors associated with interactive UI elements such as buttons, while many behaviors in the app can be triggered without user interaction.

In our work, without relying on the trustworthiness of benign apps or restricting the analysis to behaviors associated with interactive UI elements, we directly extract all user-related operations from an app and compare them semantically with the associated UI information (if no associated UI information is found, the comparison is made with all text information in the app), revealing any inconsistencies. We then present these inconsistencies to users in natural language, highlighting potential risk behaviors, thereby enabling even non-technical individuals to comprehend these behaviors.

B. Code Behavior Interpretation

Understanding code can be challenging for ordinary people. Hence, recent efforts have aimed to present code behavior in natural language. Many studies employ deep learning models to generate comments for code [41–43]. However, these comments often focus on how to use the code, rather than providing the abstract functionality that normal users seek. Additionally, most available training data is developer-oriented, rather than user-oriented. In the context of explaining malicious behavior, $XMAL$ [38] seeks to identify the factors that lead to categorizing an Android app as malicious and articulate these reasons in natural language. However, the explanation process relies on manually creating a blacklist of APIs and providing natural language descriptions for these APIs.

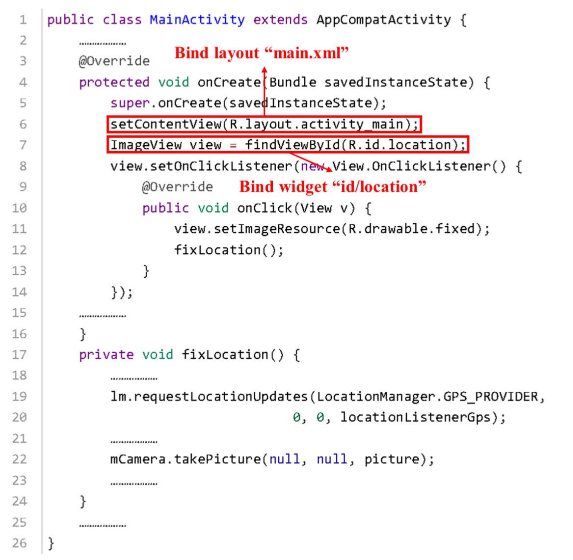
Recently, with the flourishing development of LLMs [44–46], many researchers have been exploring the application of LLMs to code summaries, yielding impressive results. For instance, Junaed et al. [16] investigate GPT-3 [47] and Codex [17] to assess their capabilities in generating code documentation, achieving excellent results with just one round of training. Toufique et al. [48] confirm similar results and demonstrate significant improvements in output using few-shot training with minimal training samples. Furthermore, Eason et al. [49] develop a Visual Studio Code plugin called GPTutor, leveraging the ChatGPT API [44], to provide concise and accurate explanations for a given source code. Therefore, in our work, we also strive to utilize LLMs to assist us in analyzing and explaining app behaviors.

III. Approach

A. Overview

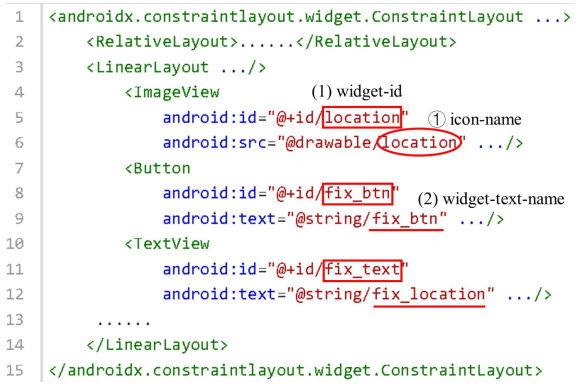
We develop a tool called *InconPreter*, which consists of four major components as illustrated in Fig. 1: the *Behavior Extractor*, responsible for extracting app behaviors; the *Behavior Filter*, which identifies behaviors related to users and devices; the *Inconsistent Behaviors Identifier*, which further finds potentially inconsistent behaviors by analyzing the consistency between code and corresponding UI elements; and the *Behavior Interpreter*, which interprets these behaviors to users in natural language and analyzes the associated risks for user reference. We will use the following example to illustrate the major components of our approach.

Example. Fig. 2 shows an example of how InconPreter extracts and interprets an inconsistent behavior. First, InconPreter utilizes static analysis to extract behaviors from the app. We define an app behavior as a sequence of API calls together with associated UI contents, e,g., *[**onCreate** (UI contents in main.xml), ..., **fixLocation** ("location", "fix_btn",*



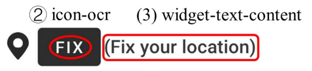
```java
1  public class MainActivity extends AppCompatActivity {
2      ..........
3      @Override
4      protected void onCreate(Bundle savedInstanceState) {
5          super.onCreate(savedInstanceState);
6          setContentView(R.layout.activity_main);
7          ImageView view = findViewById(R.id.location);
8          view.setOnClickListener(new View.OnClickListener() {
9              @Override
10             public void onClick(View v) {
11                 view.setImageResource(R.drawable.fixed);
12                 fixLocation();
13             }
14         });
15     ..........
16     }
17     private void fixLocation() {
18         ..........
19         lm.requestLocationUpdates(LocationManager.GPS_PROVIDER,
20                                   0, 0, locationListenerGps);
21         ..........
22         mCamera.takePicture(null, null, picture);
23         ..........
24     }
25     ..........
26 }
```

(a) A code example (MainActivity.java)

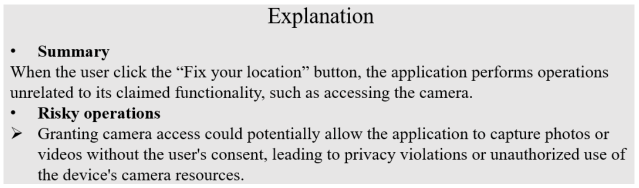


```xml
<androidx.constraintlayout.widget.ConstraintLayout ...>
    <RelativeLayout>......</RelativeLayout>
    <LinearLayout .../>
        <ImageView
            android:id="@+id/location"
            android:src="@drawable/location" .../>
        <Button
            android:id="@+id/fix_btn"
            android:text="@string/fix_btn" .../>
        <TextView
            android:id="@+id/fix_text"
            android:text="@string/fix_location" .../>
    ......
    </LinearLayout>
</androidx.constraintlayout.widget.ConstraintLayout>
```

(b) UI layout file (main.xml).



(c) Rendered UI (main.xml).



Explanation

- **Summary**

When the user click the “Fix your location” button, the application performs operations unrelated to its claimed functionality, such as accessing the camera.

- **Risky operations**

➢ Granting camera access could potentially allow the application to capture photos or videos without the user's consent, leading to privacy violations or unauthorized use of the device's camera resources.

(d) Explanation for the inconsistent behavior.

Fig. 2: An example of inconsistent behavior: (a) is a snippet of code from an app, from which a call sequence can be extracted; (b) and (c) are the associated UI elements, including the content highlighted in red in (b) and the text and icon embedded in (c). By performing the semantic comparison, the inconsistent behavior **mCamera.takePicture** (line 22 in (a)) is identified, and an explanation is given in (d).

*...), **mCamera.takePicture** (“location”, “fix_btn”, ...)]* in Fig. 2, and each API in the sequence represents an operation.

Second, apps can exhibit numerous behaviors, and many APIs are unrelated to user data or device operations, such as **onCreate** and **setContentView** in the example. We created an

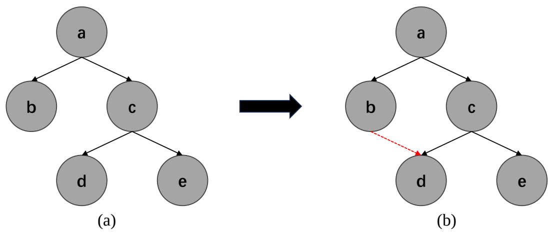


Fig. 3: An example that extends a call graph using data flow. The red dashed arrow is added according to the data flow, and the call graph is then expanded into sequences “a→b→d” and “a→c→e”, without “a→c→d”.

Attention Library to filter out these APIs, retaining only those relevant to users. This process results in *attention behaviors* like *[**lm.requestLocationUpdates** (“location”, “fix_btn”, ...)]* and *[**mCamera.takePicture** (“location”, “fix_btn”, ...)]*.

Third, InconPreter identifies inconsistent behaviors by comparing the semantics of APIs with the corresponding UI information. For the two attention behaviors mentioned above, the keyword “location” appears in the API **lm.requestLocationUpdates**, aligning with the UI content. In contrast, the API **mCamera.takePicture** includes the keyword “camera”, which is absent from the UI information, leading to its classification as an inconsistent behavior.

Fourth, InconPreter utilizes a *link graph* to map APIs within a behavior to the corresponding permissions, facilitating the analysis and summaries of remaining sensitive behaviors [18–20]. It then employs large language models (LLMs) to generate natural language interpretations of these sensitive behaviors, enabling users to understand them and identify inconsistencies. Considering that users may have limited knowledge of security and privacy, InconPreter provides a risk analysis for reference, accompanied by easily digestible explanations. For example, in Fig. 2d, InconPreter determines the API **mCamera.takePicture** accesses sensitive permission **CAMERA** by using *link graph* and provides a natural language description and risk analysis of this behavior. It informs users that when the “fix your location” button is clicked, the app operates (i.e., accessing the camera) unrelated to its claimed functionality.

In the following subsections, we will explain the details of each major component in our approach.

B. Behavior Extraction

The behavior extraction includes three parts: API sequences extraction, UI content extraction, and API-UI association.

API sequences extraction. Currently, the common method for obtaining API call sequences involves utilizing Soot [50], a Java optimization framework, to extract the call graph of the app, and subsequently expanding the call graph into multiple call sequences via depth-first traversal. For example, the call graph in Fig. 3a can be expanded into “a→b, a→c→d, a→c→e”.

However, call sequences obtained above may be incomplete and fail to represent a complete behavior. First, Soot overlooks some implicit calling relationships when constructing the

TABLE I: The extended implicit call

Caller	Callee
Thread.start	Thread.run Runnable.run
Handler.post	Runnable.run
Handler.sendMessage	Handler.handleMessage
Activity.runOnUiThread	Runnable.run
AsyncTask.execute	doInBackground onPreExecute onPostExecute

call graph, e.g., the subsequent call **Thread.run** after the invocation of **Thread.start**. Second, the above call sequences disregard logical relationships between APIs when handling data. Different operations may manipulate the same dataset (for example, "b" represents data collection and "d" represents uploading this data), implying that these operations collectively execute a single complete behavior.

To address the first issue, we analyzed and summarized common implicit calling relationships triggered by widgets linking APIs and UIs (shown in Table I), and add edges between these APIs in the call graph. To address the second issue, InconPreter leverages FlowDroid [21] to conduct data flow analysis and extend the call graph. Specifically, as shown by the red dashed line in Fig. 3b, if a function (i.e., "d") in a data flow path (i.e., "b→d") is not on the same call sequence as the source function (i.e., "b"), we add a special edge from the source function to that function. When expanding the call graph, the API nodes pointed to by the special edge are not added to the sequence of previous ordinary call relationships to avoid repeated counting of data operations represented by these nodes in subsequent analysis, i.e., the call graph is expanded into "a→b→d" and "a→c→e", without "a→c→d".

UI content extraction. In Android apps, the content displayed on the UI is mostly defined in XML files (e.g., *main.xml* in Fig. 2b), which can be directly accessed after disassembling an Android APK using Apktool [51]. Each XML file consists of widgets (e.g., *ImageView* in Fig. 2b), organized hierarchically to create what users see and interact with. The widgets mainly consist of two types: text and icons. As highlighted in Fig. 2b and Fig. 2c, for text, InconPreter extracts *widget-id*, *widget-text-name*, and *widget-text-content*, which is text content of *widget-text-name* obtained from *string.xml* file. For icons, InconPreter extracts not only *icon-name* defined in XML file, but also *icon-ocr* (the text embedded in the icon, extracted using the optical character recognition (OCR) technique [52]) and *icon-prediction* (the functional meaning of the icon, predicted by an icon classification model in Guibat [15]). *Widget-text-content*, *icon-ocr*, and *icon-prediction* are all directly presented to users in the UI, ensuring that InconPreter can extract semantically meaningful UI information.

Considering that the information on the interface and the naming of widgets may contain abbreviations or non-English languages, we utilized two well-known online abbreviation databases [53, 54] to expand common abbreviations and translated all languages into English using standardized terminology to facilitate subsequent analysis.

API-UI association. InconPreter links UI widgets to the call graph to form app behaviors. Firstly, we locate the UI layout linked to the call graph. In Android, APIs such as **setContentView** and **inflate** are commonly used to load a UI layout. For example, in Fig. 2a, the UI layout $R.layout.activity.main$ (i.e., *main.xml*) is loaded using **setContentView** at line 6. We search for such APIs in the code, where the parameter corresponds to the associated UI layout.

Secondly, we search for widgets associated with the call graph. In Android, widgets are primarily linked to code in two ways. One is through APIs such as **findViewByID** (e.g., the widget "id/location" is bound at line 7 in Fig. 2a). The other is by defining the **android:onclick** attribute in XML files, which binds a widget to the corresponding API (e.g., the *<Button android:onClick="dialPhoneMethod"/>* definition binds the button to API *dialPhoneMethod*). By searching for these patterns, we can find widgets associated with each node in the call graph. When expanding the call graph into call sequences, if a node lacks associated UI information, it inherits the UI information from its parent.

In an app, many behaviors are not bound to any widgets, and they can run in the background without requiring user interaction. If the app does not display any information on its UIs to inform the user about such behaviors, this poses a significant risk, regarded as *self-starting inconsistent behaviors*. For a self-starting behavior, the root node of the corresponding call graph lacks any UI information, and thus we consider all UI information extracted from the app as its UI information.

C. Behavior Filtering

Behavior Filtering filters out behaviors irrelevant to users from all the behaviors extracted. Considering that Android apps can access and modify sensitive data, system resources, and device states exclusively through the utilization of corresponding Android system APIs [55], InconPreter only retains Android system APIs documented in the official Android document [56] within each behavior. For example, in Fig. 2a, we retain system APIs such as **mCamera.takePicture**, and filter out other APIs such as **onCreate**. Focusing solely on Android system API call sequences also helps InconPreter mitigate the challenges of code obfuscation, as most apps do not obfuscate system APIs due to the risk of causing system crashes.

InconPreter then conducts thorough filtering of the behaviors based on an *Attention Library*, which is a repository of keywords that are likely to be of interest to users. We design a three-step filtering method to establish the Attention Library, with which InconPreter employs a keyword matching method for the thorough filtering. Specifically, if none of the keywords in an API are present in the Attention Library, the API will be filtered out, and the remaining APIs form the *attention behaviors*. Note that the filtering is based on individual keywords rather than entire APIs because APIs may change with the updating of Android versions, but the keywords constituting these APIs remain relatively stable. By constructing a lexicon instead of an API library, InconPreter can be more adaptable to continuously updating versions.

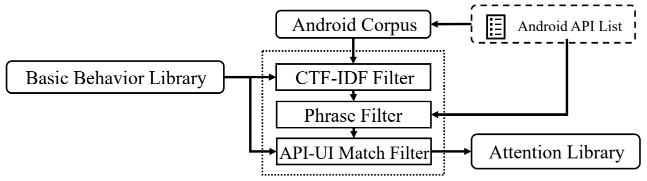


Fig. 4: The framework of three-step filtering

The process of the three-step filtering to establish the Attention Library is shown in Fig. 4. We first collect a de-duplicated *Android API List* from the Android API documentation [56], across versions spanning from Android 4.0 to Android 14.0. Then we tokenize the APIs using the CamelCase split method [57] (e.g., *takingPicture* is tokenized into "taking", "Picture"), and each token is converted to its lemma using NLTK [58] (e.g., "taking" is converted to "take") to form the initial *Android Corpus*. Next, we designed a three-step filtering method based on some observations to select keywords from the Android Corpus that are likely related to behaviors of interest to users, forming the Attention Library.

Observation 1: "Unimportant" words appear more frequently in various apps than those related to sensitive resources.

We collect behaviors from both benign and malicious apps, forming the *Basic Behavior Library* (short for *BBL*). The benign apps are randomly selected from Google Play, and the malicious apps are from the CIC malware dataset [59]. Then, we statistically analyze the frequency of each word in the *Android Corpus* across the APIs in the *BBL*. We found the words about sensitive resources utilized in previous works [12, 13, 15] (e.g., "contact", "camera", "location") all exhibit relatively moderate probabilities. In contrast, words like "string", "builder", and "view" have higher frequency.

CTF-IDF Filter. Based on this observation, we designed a CTF-IDF (Corpus-based Term Frequency-Inverse Document Frequency) filter. CTF-IDF adapts the principles of TF-IDF [60] to evaluate the overall frequency of a term across all apps. If a term has a high frequency among all the terms and appears in most apps, it is considered unimportant and should be filtered out. CFT-IDF is defined as below:

$$ctf - idf = ctf \cdot idf \quad (1)$$

where ctf represents the frequency of a term t within the corpus *BBL*, and idf assesses the distribution of a term across different apps. The formulas are as follows:

$$ctf = \frac{n}{N} \quad (1a)$$

$$idf = log\frac{M}{1 + |\{j : t \in d_j\}|} \quad (1b)$$

where n is the times t appears in *BBL*, N is the total number of terms in *BBL*, M is the number of apps, and $|\{j : t \in d_j\}|$ denotes the number of apps in which the term t appears.

Observation 2: Words that combine with many other words are not important.

Android API developers typically adhere to certain naming conventions, such as naming an API based on the action they perform on specific objects (e.g., **mCamera.takePicture**), so that programmers can easily understand the functionality of the API. However, there are common verbs (e.g., "get", "set", and "load") and nouns (e.g.,"data") frequently appearing in API names combined with other words. These words are user-irrelevant, thus needing to be filtered out.

Phrase Filter. We use the Android API List as our database and analyze the phrase patterns of each word. Specifically, we first conduct dependency syntactic parsing implemented by spaCy [61] on the name of each API (the class name and function name parsed separately) in the Android API List to obtain the part-of-speech tags of each word in the name and the relationships between words. Then we count the number of nouns combined with each verb and the number of verbs combined with each noun. If the count exceeds a predefined threshold, the word will be removed from Android Corpus.

Observation 3: An API should be important when it is semantically related to its UI elements.

UI represents the behavior that the app informs the user it will perform, while the APIs corresponding to the UI elements reflect the actual behavior executed by the app. When the API and its UI information exhibit semantic correlation, it indicates that this semantic information is important to convey to the user, and its semantic words are significant.

API-UI Match Filter. After the first two filtering steps, we perform a semantic comparison between API and UI content for each node in the API sequences in *BBL*. If there is a semantic match (i.e., at least one keyword in the UI content matches a keyword in the API), and the matching keyword is present in the Android Corpus, the keyword is added to the Attention Library. Although the initial size of the Attention Library is influenced by the apps within *BBL*, this library can be continually expanded with data from future test apps.

D. Inconsistent Behavior Identification

The Inconsistent Behavior Identifier performs consistency analysis by comparing the *semantic keywords* of APIs and UI content in the attention behaviors. The semantic keywords refer to keywords in APIs or UI contents, which exist in the Attention Library, such as "camera", "picture" in the API **mCamera.takePicture** and "location" in the UI *main.xml* in Fig. 2. For a semantic keyword of an API, if its associated UI contents do not contain any synonyms of that keyword, the API operation will be flagged as UI-unmatched, and the entire attention behavior will be marked as an inconsistent behavior. For example, in Fig. 2, the semantic keywords of API **mCamera.takePicture** (i.e., "camera" and "picture") do not match the keywords in the associated UI contents (i.e., "location"), so this attention behavior is marked as inconsistent.

E. Behavior Interpretation

Due to the complexity of API sequences, which can be difficult for average users to understand, InconPreter incorporates a Behavior Interpreter, which translates the app's behavior

TABLE II: The design of prompts for behavior explanation and risk analysis

Type	Function	Example
role_prompt	Assigning roles to LLM so it can focus more on specific domains and access relevant knowledge accordingly.	You are an expert in mobile security, skilled in analyzing behaviors of Android apps and identifying potential risks.
background_prompt	Introduce the relevant background knowledge of inconsistent behavior.	Mobile apps are developing rapidly, but we have found inconsistencies between the claimed functionality and the actual operations in some mobile apps, which threatens user data security. Note that if the operations executed by the app do not reflect the current UI or app functionality, it is considered an inconsistency.
intro_prompt	Introduce the app and the discovered inconsistent behaviors.	In an app categorized as <*app category*>, the claimed functionality (or user-performed action) is <*UI sematics*>. During this process, we found inconsistent operations: <*permissions requested*>.
prepare_prompt	Inform LLM that the following are tasks to be performed so that it better understands what needs to be done.	You have the following two tasks.
task_prompt 1	Give the first task: summarize the inconsistent behaviors of the app in simple and user-friendly language.	Explain in plain language what the inconsistencies are. Output format: When users use <xxx> app (or perform a <xxx> action), the app performs <xxx>.
task_prompt 2	Give the second task: analyze the risky operations among the inconsistent behaviors and provide explanations for the analysis.	Among these inconsistent operations (<permissions requested>), which ones are necessary to achieve the desired functionality, and which ones may cause potential risks to users or devices? Give explanations for each operation. Output format: Necessary operation: [<*permission xxx*>, <*permission xxx*>]; Risky operation: [<*permission xxx*>, <*permission xxx*>]; Explanation: <xxx>.

into natural language, helping users understand the app's inconsistent behaviors and alerting them to potential risks.

For interpretation, we first find permissions associated with the APIs to identify user-related data operations. In Android, accessing user data or system resources requires APIs associated with relevant permissions [18–20]. For example, the API **android.location.Criteria: void setAltitudeRequired** requires the **ACCESS FINE LOCATION** permission. Therefore, we can infer the operation of this API using the meaning of this permission, i.e., accessing precise location information from the device, including GPS, etc. If an API is not associated with any permissions, it implies that it is unrelated to sensitive resources, and we do not need to interpret it. We construct a *Link Graph* that maps APIs to their corresponding permissions. This mapping is crawled from the official Android documentation [56], and we also leverage the mapping list from the previous work PScout [62] to extend the Link Graph.

With the *Link Graph*, we can obtain all the permissions requested by a behavior. However, if we express the semantics of the actions solely using these specialized terms (e.g., **ACCESS NETWORK STATE**, **READ PHONE STATE**), ordinary users may still find it difficult to understand what they represent. Considering the outstanding performance of large language models (LLMs) in semantic understanding and language expression [63–65], we apply them to generate simpler and more user-friendly explanations for each behavior. In addition, we provide an analysis of the potential risks to users for reference in case users may have limited knowledge of security and privacy.

Specifically, we employ prompt engineering techniques and design appropriate prompts to stimulate the ability of large language models for our task. The design of our prompts is present in Table II. For each inconsistent behavior, we fill in the relevant information extracted earlier into the corresponding positions in the *intro prompt* and sequentially input these prompts to the LLM using its API calls. Then, the LLM will generate an explanation and risk analysis report for the inconsistent behavior according to the output templates specified in the *task_prompt*, as exemplified by Fig. 2d. With these explanations, users can understand the behaviors by themselves and discern any risky inconsistent behaviors.

IV. Evaluation and Findings

A. Experiment Setup

Dataset and Experiment. We primarily evaluate the performance of InconPreter on the following datasets.

Wild apps. We collected 10,878 apps across all 34 categories from Google Play based on their popularity between 2020 and 2024, with approximately the same number of apps in each category[1]. Wild apps serve as the source for evaluating InconPreter's inconsistent behavior detection and interpretation performance and help us gain a deeper understanding of the inconsistent behaviors through the results.

Comparison dataset. To evaluate InconPreter's accuracy in extracting inconsistent behaviors and compare the results with DeepIntent [13], the most relevant current work, we collected a comparison dataset consisting of 600 apps, including 510 wild apps and 90 samples provided by DeepIntent.

Malware dataset. To further evaluate InconPreter's performance in identifying risky behaviors, we collected a malware dataset comprising 100 randomly selected samples from the widely used AMD (Android Malware Dataset) [66], and used

[1]Use *https://app.diandian.com/rank/googleplay* for categorizing and ranking.

the AMD reports (detailed human-generated reports for each malicious sample) and VirusTotal reports [67] as references.

Preparation. To use InconPreter, it is essential to first construct the Attention Library and the Link Graph. We spent approximately 2 weeks building the Attention Library following the steps outlined in III-C, using the randomly selected 500 wild apps and 500 malicious apps from the CIC dataset [59]. We spent several hours crawling the Android official documentation across versions from Android 4.0 to Android 14.0 to gather the API-permission mappings, which, along with the results from PScout [62], formed the Link Graph.

Platform and Runtime. It took approximately 4 months to analyze all wild apps (about 16 minutes per app) on servers equipped with an Intel(R) Xeon(R) CPU E7-4830 and 188GB of RAM. If more computing resources were available to support parallel processing, the time could be further reduced.

B. Effectiveness of Inconsistency Extraction

Since there is no benchmark dataset in this field yet, we compare InconPreter with the most related work, DeepIntent. DeepIntent associates each widget on the UI with executed APIs and their corresponding permissions. It then uses a model to predict the behavior of this widget and determines whether the executed API requests additional permissions, indicating inconsistent behavior. DeepIntent outputs inconsistent behaviors as <*widget-id, API, permission*> tuples, which we can use as the comparison target. However, due to methodological differences between InconPreter and DeepIntent, for a given *widget-id*, we might identify the same API mapped to different permissions, or the same permission mapped from different APIs. Since both API and permission in the <*widget-id, API, permission*> tuple represent the app's operation associated with the current widget-id, we consider the behavior the same if either the API or the permission matches. Specifically, for DeepIntent and InconPreter, we analyze their result tuples to merge all <*widget-id, API, permission*> tuples into new <*widget-id, permission*> tuples and firstly compare the permissions. With the same *widget-id*, given two tuples from two tools, if the permissions match, it's considered a match. If not, we then check whether DeepIntent and InconPreter outputs the same <*widget-id, API*>. If they do, it is considered a match. However, this may lead to some duplicate counting. For example, for DeepIntent's tuple <*w1, API1, permission1*>, InconPreter might count <*w1, API1, permission1*> as a match due to *permission1* matching, and then count <*w1, API1, permission2*> again due to *API1* matching. To avoid duplicate counting, we will not match *API1* in the second comparison if it has been associated with *permission1* in the permission matching phase.

On the 600 apps in the comparison dataset, DeepIntent extracts a total of 1,026 risky inconsistent behaviors, while InconPreter extracts 1,625. After manually checking the results, we find 102 false positives (i.e., the extracted behaviors are not actual inconsistencies) in DeepIntent's results, and 83 false positives in InconPreter's results. Therefore, the precision of InconPreter in identifying risky inconsistent behaviors is 94.89%, better than DeepIntent's 90.06%. We analyze the

TABLE III: Comparision with DeepIntent in inconsistent behaviors extraction

Common	**InconPreter Only**		**DeepIntent Only**
	with widget	without widget	
838	280	424	86

reasons behind the false positives generated by InconPreter. The main cause is the inaccurate extraction of relevant UI content bound to the API during static analysis.

After removing false positives, we compare our results with DeepIntent's. As shown in Table III, DeepIntent extracts 924 risky inconsistent behaviors, and we identify 1,542 behaviors, of which 838 are common. After analyzing the 86 behaviors that InconPreter missed, we find that some share common causes. 21 of the missed behaviors are **VIBRATE** and **WAKE LOCK**, which may disturb users and accelerate battery consumption. InconPreter incorrectly regards them as functionally necessary in some music or communication apps, and therefore does not mark them as risky. 13 behaviors are related to the API **android.app.ActivityManager: void killBackgroundProcesses**, i.e., **RESTART PACKAGES** and **KILL BACKGROUND PROCESSES**. We primarily focus on actions directly related to user and device data, while these behaviors pertain mainly to impacts on the app process, which falls outside our scope.

While we miss 86 inconsistent behaviors, we find 280+424 inconsistent behaviors missed by DeepIntent. Specifically, we discover an additional 280 inconsistent behaviors that are bound to widgets, such as reading SMS and contacts, and installing another app when users click a "video_start" button. These behaviors can potentially have a significant impact on users. Furthermore, we can identify 424 self-starting inconsistent behaviors that are not associated with any widgets. For instance, an app *wabao.ETAppLock* quietly accesses the device status and identity information, reading the contacts and other sensitive data in the background. Another app *com.apostek.untangle* access the device's camera, allowing the ability to take photos or record videos and get the user location when the user merely opens the app without any additional interaction.

Our superior performance may be attributed to the fact that DeepIntent is trained based on a dataset that collects mappings between icons and permissions, overlooking the differences in the harmfulness of these mappings across different apps. That is, behaviors may be benign in some apps or contexts but risky in others. In addition, DeepIntent can only discover inconsistencies related to behaviors associated with interactive UI elements such as buttons, while many behaviors in the app can be triggered without interaction. In our study, for each app, InconPreter first extracts potentially sensitive operations related to user data or device resources. Then, it investigates whether the app's execution of these behaviors aligns with the semantics communicated to users, including both fine-grained semantics (i.e., semantics associated with the widgets bound to APIs) and coarse-grained semantics (i.e., semantics conveyed by all UI elements within the app). This allows us to discover more inconsistencies, including some behaviors that DeepIntent cannot detect, such as apps secretly accessing users' locations while chatting, or accessing the network to consume mobile data while watching offline videos maliciously or mistakenly.

Additionally, we recognize that not all semantically inconsistent behaviors pose risks. For instance, when a user employs a gallery app to search for images, the app may access the network or read resources locally. While these actions may not be evident in the UI, they are reasonable for such a task. Therefore, we further analyze these inconsistent behaviors from a functional perspective. However, DeepIntent does not consider this aspect, resulting in false positives.

C. Validation of Risks Analysis in Behavior Interpretation

InconPreter provides an analysis of potential risks within behaviors to users as a reference. To avoid misleading users, the analysis should be reasonably accurate. Therefore, we validate the performance of the risk analysis on the malware dataset.

AMD and VirusTotal reports provide generalized summaries of behaviors, often combining multiple risky behaviors into a single sentence. For comparison, we first need to break down the behaviors into *basic sub-behaviors*. We define a *basic sub-behavior* as a sensitive operation corresponding to a single permission. For example, the behavior "Send GPS to a remote location" can be divided into "Access precise location" (**ACCESS FINE LOCATION**) and "Open network sockets" (**INTERNET**). We manually analyze all sentences in the reports to extract the sub-behaviors and compare them with our results.

For the 100 malicious samples, AMD and VirusTotal report a total of 855 *basic sub-behaviors*. InconPreter successfully identifies 782 (91.46%) of these behaviors. Additionally, we discover 27 new risky behaviors from 21 apps, including **ACCESS FINE LOCATION**, **READ PHONE STATE**, etc. These risks are not included in the reports from AMD and VirusTotal, but they do pose a threat to user privacy.

For the 73 malicious behaviors not identified, we find that 24 missed behaviors involve APIs with generic keywords not present in the *Attention Library* (e.g., the behavior "Read database like contact or SMS" corresponds to the "ContentResolver.query()" API with keywords "content", "resolver" and "query") and are filtered out. 21 behaviors are related to network activities, where static analysis cannot fully discern the intent of network requests or data transmissions, resulting in potential misjudgments of malicious intent. For the remaining 28 behaviors, we do extract the relative APIs, but the lack of corresponding permissions in the manifest file indicates the app does not actually have permission to execute these behaviors. The AMD report also does not include these behaviors, which could be false positives from VirusTotal. Therefore, the risk identification rate of InconPreter should be 94.56%.

D. Human Perception of Behavior Interpretation

To demonstrate the effectiveness of InconPreter in helping users understand app behaviors and recognize risky behaviors, as well as to understand user opinions on the identified risky behaviors, we conduct survey studies. In the first questionnaire, participants are shown 7 UIs from apps with different functionalities, along with explanations for the detected inconsistent behaviors. Participants rate the explanations in terms of comprehensibility and reasonability. We also inquire about their perceptions of the behaviors and whether our explanations increase their awareness of the risks. In the second questionnaire, participants rate their levels of concern about each risky behavior. Both questionnaires were reviewed and received IRB approval at our institute. The main questions are illustrated in the Appendix.

We distributed the questionnaires online and recruited participants via social media. Before consenting, participants were informed of the survey's purpose, procedures, risks, benefits, and lottery-based compensation. We received a total of 256 valid responses. 14 were excluded due to unfinished answers, inconsistent age and occupation, and repeated selections. All participants are anonymous, and the collected data does not identify any individuals and is securely stored.

The participants are from different fields and age groups. This allows us to explore how individuals from diverse backgrounds perceive and understand our analysis, thereby better helping them mitigate risks. The respondents range in age from 18 to 41 years, with an average age of 22. Among the 256 respondents, 205 (80.08%) either hold or are pursuing a bachelor's degree or higher, and 66 (25.78%) have a background in computer-related fields. The results indicate that our explanations are easily understood by the vast majority of respondents, with an average score of 4.07 (on a scale of 1-5, where 5 indicates easiest to understand). Even participants with a high school education or below and those outside the computer field achieve an average score of 4.04. Furthermore, the analysis of risky inconsistent behaviors in our explanations receives recognition from participants, with an average score of 4.15 (on a scale of 1-5, where 5 indicates the most reasonable). Participants with a background in computer science also rate this aspect highly, with an average score of 4.11.

In the first questionnaire, for the risky behaviors we identified, on average, 25.45% of participants claim they do not notice the risks before. However, their level of concern regarding these risks is high, with an average score of 4.03 (on a scale of 1-5, where 5 indicates the most concerned). In the second questionnaire, regarding the 32 identified risks, users rate their level of concern on average as 4.01. The highest concern is for **CAMERA** (accessing the device's camera, allowing the ability to take photos or record videos), with an average score of 4.23, while the lowest concern is for **WAKE LOCK** (preventing the device from entering sleep mode to ensure it stays awake during specific operations, which accelerates battery and resource consumption), with an average score of 3.81. These results suggest that such inconsistent issues in mobile apps are quite serious. Despite users' high concern for their privacy and device security, they may lack awareness of relevant knowledge in their day-to-day experience.

With the help of our explanations for app behaviors, participants are able to better understand issues within the apps. Overall, participants find our explanations very helpful, with an average score of 4.12 (on a scale of 1-5, where 5 indicates most helpful). Particularly for individuals outside the

TABLE IV: Categories of apps containing risky inconsistent behaviors

Category	Communication	Education	Entertainment	Finance	Game	Fitness	Life & Traveling	Reading	Office	Gallery	Photography & Beauty	Tools	Video & Audio	Total
app num	29	29	26	11	***59***	14	43	42	24	20	23	***59***	34	**413**
risks	206	76	161	28	156	41	102	158	154	51	82	313	136	**1664**
risks per app	***7.10***	2.62	***6.19***	2.55	2.64	2.93	2.37	3.76	***6.42***	2.55	3.57	***5.31***	4.00	**4.03**

TABLE V: Performance comparison between different LLMs.

	TP	FP	TN	FN	Precision	Recall	Accuracy
GPT-4	233	18	201	13	92.83%	94.72%	93.33%
GPT-3.5	214	61	158	32	77.82%	86.99%	80.00%
Llama-2	66	38	181	180	63.46%	26.83%	53.12%

computer science field, the assistance is more pronounced, as they gave an average score of 4.26.

E. Performance Comparison between LLMs

In the Behavior Interpreter, we employ LLM to provide natural language descriptions of detected inconsistent behaviors and analyze the risks involved. As LLM technology is gaining momentum, numerous high-performing LLM models have emerged. We select several popular LLMs (GPT-3.5, GPT-4 and Llama-2) to compare their performance for our task.

Given the perfect performance of LLMs in language representation, we mainly compare their performance in analyzing risky behaviors. We randomly select 100 apps from the wild apps that contain 465 inconsistent behaviors in total. These behaviors are manually annotated to determine if they constitute risky behaviors, i.e., if they truly violate the functionality claimed by the app. We recruit 3 volunteers with at least three years of experience in Android development or security analysis. They individually analyze each inconsistent behavior to determine if it is a risky behavior. In cases of disagreement, they vote to arrive at a consensus. Ultimately, they identify a total of 246 risky inconsistent behaviors.

Afterward, we evaluate the 465 inconsistent behaviors using three popular LLMs. The results are shown in Table V. From the results, we observe that without modifying the LLMs, GPT-4 achieves the best performance in our task, with a recall of 94.72% and an accuracy of 93.33%. This means that most risky behaviors can be accurately identified. On the other hand, Llama-2 performs poorly in this task, with an accuracy of only 53.12% and a recall of only 26.83%. Therefore, InconPreter utilizes GPT-4 for the risk analysis task.

F. Findings in The Wild

To gain a deeper understanding of inconsistent behaviors in apps and draw users' attention to these issues, we applied InconPreter to the 10,878 wild apps to explore the risky inconsistent behaviors they contain. For ease of counting, we consider using a single permission to represent each risky behavior. The behaviors are illustrated in Table X in the appendix. We identify a total of 1,664 risky inconsistent behaviors among 413 (3.80%) of these apps, with an average of approximately 4 risky behaviors per app, as depicted in Table IV. These 413 apps cover all categories and have been downloaded over 4.3 billion times collectively, among which, 89 (21.55%) apps have downloads exceeding 1 million for each. The average rating of these apps is 3.97, with 122 (29.54%) of them having a rating exceeding 4.0 out of 5. This indicates that the risks we discovered are widespread and have already affected millions of mobile users.

To gain a deeper understanding of inconsistent behaviors, raise awareness among users about app inconsistencies, and improve their consciousness of protecting their privacy and device security, we explore the following questions:

Q1: What are the main risky behaviors? Which risky behaviors occur more commonly?

We list the risky behaviors in Table VI and count the number of apps containing each type of risky behavior. We find that READ PHONE STATE, ACCESS NETWORK STATE, WAKE LOCK, and ACCESS COARSE LOCATION are the most common risky behaviors. Additionally, we discover risky behaviors related to personal data in certain apps. For example, 93 apps contain ACCESS FINE LOCATION, posing risks of location tracking. 6 apps request device identifiers for unauthorized authentication. 25 apps secretly access account information stored on the device, including local accounts and those synced with the device, such as Google accounts, social media accounts, and email accounts. 4 apps initiate unauthorized phone calls, and another 5 conduct unauthorized audio recordings. Some apps directly access and manipulate contact data on the device, and some apps receive and read user SMS messages, with some even sending unauthorized SMS messages. These behaviors significantly threaten user and device privacy and security.

Q2: Which categories are more likely to contain risky inconsistent behaviors?

In Table IV, we count the number of apps containing risky inconsistent behaviors and the total number of risky inconsistent behaviors within each category. To facilitate the statistics, we compress the 34 categories provided by Google Play into 13 categories based on the apps' functionality.

We discover the highest number of apps that contain inconsistent risk behaviors in the categories “Game” and “Tools”, with each category containing nearly 60 apps. This could be because, in the case of “Game”, users are often deeply immersed in gameplay, making it easier for apps to perform additional actions without being easily noticed. In “Tools” apps, they often require direct interaction with the device's system, leading to a higher likelihood of requesting various permissions and potentially engaging in risky inconsistent behaviors.

Additionally, we compare the average number of risky inconsistent behaviors per app across each category. In the

TABLE VI: Distributions of risky inconsistent behaviors

Category / **Behavior**	Communication	Education	Entertainment	Finance	**Game**	Fitness	Life & Traveling	Reading	Office	Gallery	Photography & Beauty	**Tools**	Video & Audio	**Total**
READ PHONE STATE	10	11	9	4	32	1	16	***22***	8	6	3	24	***24***	170
ACCESS NETWORK STATE	10	8	11	2	22	6	11	14	4	6	13	22	12	141
WAKE LOCK	12	13	7	5	5	5	16	16	8	6	7	13	9	122
ACCESS COARSE LOCATION	9	9	8	4	13	5	10	9	7	5	9	13	***12***	113
ACCESS FINE LOCATION	9	5	6	4	5	6	7	9	10	5	7	14	6	93
WRITE EXTERNAL STORAGE	4	6	***8***	3	8	5	1	8	5	1	3	18	6	76
ACCESS WIFI STATE	3	3	1	0	9	2	9	5	3	3	3	11	3	55
INTERNET	0	0	3	0	2	3	2	0	3	***8***	***9***	17	4	51
VIBRATE	2	1	4	0	3	1	4	**4**	2	2	1	0	3	27
READ EXTERNAL STORAGE	1	0	***5***	1	1	1	2	1	2	0	5	6	2	27
GET ACCOUNTS	2	2	0	1	3	1	2	**4**	4	0	0	5	1	25
SEND SMS	2	0	2	0	7	0	0	1	1	1	0	7	2	23
BLUETOOTH	3	2	1	0	1	1	2	1	2	0	0	4	3	20
READ CONTACTS	3	1	1	1	1	0	2	1	3	0	1	4	0	18
WRITE SETTINGS	2	0	1	0	1	0	0	1	0	1	0	8	3	17
CAMERA	0	3	0	0	0	1	**5**	0	3	0	0	2	0	14
CHANGE WIFI STATE	1	0	0	0	2	0	1	1	0	0	0	8	0	13
READ SMS	2	0	0	0	2	0	0	0	1	1	0	3	1	10
SYSTEM ALERT WINDOW	0	1	1	0	0	1	1	0	1	0	1	2	1	9
BROADCAST STICKY	2	1	1	0	0	0	0	0	2	0	0	0	1	7
MODIFY AUDIO SETTINGS	1	4	0	0	0	0	1	0	0	0	0	1	0	7
SCHEDULE EXACT ALARM	2	2	1	0	0	0	0	1	0	0	0	1	0	7
USE ICC AUTH WITH DEVICE IDENTIFIER	1	0	0	0	2	1	1	0	0	0	0	1	0	6
RECEIVE SMS	1	0	0	0	0	0	0	0	2	0	1	1	0	5
RECORD AUDIO	0	0	0	0	0	0	1	1	3	0	0	0	0	5
CALL PHONE	1	0	0	0	0	0	1	0	0	1	0	1	0	4
CHANGE NETWORK STATE	1	0	0	0	0	0	1	1	0	0	0	1	0	4
WRITE CONTACTS	1	0	0	0	0	0	0	0	1	0	0	1	0	3
INSTALL PACKAGES	1	0	0	0	1	0	0	0	0	0	0	0	0	2
NEARBY WIFI DEVICES	0	0	0	0	0	0	0	0	0	0	0	2	0	2
BLUETOOTH SCAN	0	0	0	0	0	0	0	0	0	0	0	1	0	1
FOREGROUND SERVICE	0	0	0	0	0	0	0	0	0	0	0	1	0	1

TABLE VII: Categories of apps containing self-starting risky inconsistent behaviors

category	Communication	Education	Entertainment	Finance	Game	Fitness	Life & Traveling	Reading	Office	Gallery	Photography & Beauty	Tools	Video & Audio	**Total**
app num (ratio)[1]	24 (82.76%)	***28*** (***96.55%***)	21 (80.77%)	10 (***90.91%***)	50 (84.75%)	8 (57.14%)	***38*** (88.37%)	***38*** (***90.48%***)	19 (79.17%)	***18*** (***90.00%***)	18 (78.26%)	23 (38.98)	27 (79.41%)	322 (77.97%)
risks	61	65	59	21	94	22	76	89	49	34	43	65	62	740 (44.47%)[2]
risks per app	2.54	2.32	2.81	2.10	1.88	2.75	2.00	2.34	2.58	1.89	2.39	2.83	2.30	2.30

[1] the percentage of apps containing self-starting risky inconsistent behaviors among all apps containing risky behaviors within each category.
[2] the percentage of self-starting risky inconsistent behaviors among all risky inconsistent behaviors.

"Communication", "Office", "Entertainment" and "Tools" categories, the average number of risky inconsistent behaviors per app is higher than others, exceeding 5, which surpasses the overall average. Particularly in the "Communication" category, the number even exceeds 7. This may be because these categories of apps have richer functionality compared to others, making it easier to hide risky behaviors within them.

Q3: Is there any correlation between risky inconsistent behaviors and app categories?

In Table VI, we count the number of apps containing each type of risky inconsistent behavior across different categories. We find that for some behaviors, there are indeed differences in the number of occurrences among different categories. For the majority of risky behaviors, the categories containing them most frequently are almost always the "Game" or "Tools". For several behaviors, more than half are found in these two categories of apps, such as **SEND SMS**, **CHANGE WIFI STATE**. This could be due to the reasons discussed in *Q1*.

Apart from these two categories, there are several interesting findings. **READ PHONE STATE** appears most frequently in the categories "Video & Audio" and "Reading", despite seeming unrelated to the primary functionality of apps in these categories. **ACCESS COARSE LOCATION** is prevalent in the category "Video & Audio". While location may be legitimately used for personalized recommendations, accessing it without user consent violates privacy norms. Behaviors involving **READ EXTERNAL STORAGE** and **WRITE EXTERNAL STORAGE** are commonly observed in the category "Entertainment". **INTERNET** is frequently requested by apps in the "Gallery" and "Photography & Beauty" categories. Some of these apps initiate internet communication even for tasks that could be performed offline, such as browsing local albums or taking photos. However, this behavior might involve uploading local image resources to the cloud, potentially compromising user privacy and consuming mobile data without users' consent.

In addition, we find that in each category, the most commonly encountered risky behaviors are generally behaviors mentioned in *Q1*. Excluding these behaviors, we observe that in the "Life & Traveling" category, **CAMERA** behavior is more prevalent.

TABLE VIII: Distributions of self-starting risky behaviors

behaviors type	app num	ratio[1]
READ PHONE STATE	124	72.94%
WAKE LOCK	111	90.98%
ACCESS NETWORK STATE	94	66.67%
ACCESS COARSE LOCATION	84	74.34%
ACCESS FINE LOCATION	60	64.52%
WRITE EXTERNAL STORAGE	52	68.42%
ACCESS WIFI STATE	34	61.82%
VIBRATE	26	96.30%
INTERNET	25	49.02%
READ EXTERNAL STORAGE	20	74.07%
BLUETOOTH	14	70.00%
GET ACCOUNTS	14	56.00%
CAMERA	10	71.43%
READ CONTACTS	10	55.56%
SEND SMS	9	39.13%
WRITE SETTINGS	7	41.18%
SYSTEM ALERT WINDOW	6	66.67%
SCHEDULE EXACT ALARM	6	85.71%
BROADCAST STICKY	5	71.43%
MODIFY AUDIO SETTINGS	5	71.43%
READ SMS	4	40.00%
CHANGE WIFI STATE	4	30.77%
CHANGE NETWORK STATE	3	75.00%
USE ICC AUTH WITH DEVICE IDENTIFIER	3	50.00%
CALL PHONE	2	50.00%
RECORD AUDIO	2	40.00%
RECEIVE SMS	2	40.00%
WRITE CONTACTS	1	33.33%
BLUETOOTH SCAN	1	100.00%
INSTALL PACKAGES	1	50.00%
NEARBY WIFI DEVICES	1	50.00%
FOREGROUND SERVICE	0	0.00%

[1] the percentage of apps containing each self-starting risky inconsistent behavior among all apps containing such risky inconsistent behavior.

VIBRATE and **GET ACCOUNTS** behaviors are more common in the "Reading" category. Additionally, "Game" and "Tools" categories contain more **ACCESS WIFI STATE** and **SEND SMS** behaviors. Hence, when users are concerned, they can pay particular attention to the risky behaviors commonly found within the category to which the app belongs.

Q4: How many self-starting risky inconsistent behaviors are there?

As shown in Table VII, We find a total of 740 self-starting risky inconsistent behaviors among 322 (77.97%) apps, accounting for 44.47% of all identified risky behaviors. These apps span across all categories. On average, each app has approximately 2 self-starting risky inconsistent behaviors. Additionally, we tabulate the number of apps containing each type of self-starting risky inconsistent behavior in Table VIII, and we find that the ranking of common self-starting risky behaviors is similar to the ranking of common risky behaviors identified earlier. We then calculate their percentages relative to all apps containing such risky behavior. We find that for 23 out of the 32 risky behaviors, more than half of the apps containing these behaviors self-start them. An interesting finding is that behaviors typically considered to require user interaction, such as **WRITE CONTACTS**, **SEND SMS**, **CALL PHONE**, can also self-start in some apps. These indicate that self-starting risky inconsistent behaviors constitute a significant percentage and are widely distributed. The results also underscore that previous work focusing solely on app behaviors related to user interaction may overlook many potential risks.

Q5: Which categories of apps are more likely to contain self-starting risky inconsistent behaviors?

We count the number of apps containing self-starting risky inconsistent behaviors for each category in Table VII. We find that the category with the highest number of apps containing such behaviors is "Game" followed by "Life & Traveling" and "Reading". Interestingly, the "Tools" category which previously ranked highest drops in rank. Additionally, we observe that in the categories "Education", "Finance", "Reading" and "Gallery", the percentage of apps containing self-starting risky inconsistent behaviors among all apps containing risky inconsistent behaviors is the highest.

Q6: Are there any typical instances of inconsistent behaviors?

In our experiments, we discover numerous instances of risky behaviors, including some interesting and noteworthy cases.

Case 1: when you're having fun, risks may be lurking. When users are using their phones for entertainment, they often overlook other activities happening on their devices. Apps may take advantage of this distraction to perform actions without the user's knowledge. For example, the app *com.banjen.app.balalaika* might secretly send a text message to a particular number while the user is engaged in playing guitar, thus compromising the user's phone number and other information. Another educational game app *com.preschoolacademy.toddlerpreschoollearning* might access the device's precise location, obtain the phone's status information, modify certain system settings, and access user accounts stored on the phone, posing a threat to user privacy and security.

Case 2: your location may have been continuously exposed. Location is one of the most valued aspects of personal privacy for users, but we find that users' locations might be leaked through various means. For example, in the app *kha.prog.mikrotik*, when users use their phones as hotspots, the app reads the device's location. Similarly, in the app *net.braincake.pixl.pixl*, the location might also be collected when users take photos. Additionally, in some media and social apps, location might be collected for features like location-based recommendations. However, these apps, e.g., *com.wtoday1usabvd* and *com.kubetho.hotroku* might continuously collect device location during runtime, even if the user is not actively engaged in actions related to location.

Case 3: Offline-use apps or functions might also consume mobile data. Currently, mobile data plans can be costly, so people are mindful of their data usage. However, we find that in some apps, when users engage in what they assume are offline features, the app still consumes network data. For example, in the video playback app *com.samsung.rms.retailagent.proxy*, the app continues to access the network when users play locally downloaded videos. As another example, in the piano playing app *com.synthesia.synthesia*, the app continues to access the network while the user is playing the piano.

TABLE IX: Evolution between different periods

	2010-2014	2015-2019	2020-2024
percentage of apps containing risks	26.44%	15.64%	3.80%
number of risky behaviors which more than 10% apps contain	22	14	8

Q7: Are there any changes over time in the primary risky inconsistent behaviors in mobile apps with the development of apps and increasing concern for privacy and security?

Since early versions of apps are no longer available for download on official markets, we manually searched and obtained 261 apps released from 2010 to 2014, and 748 apps released from 2015 to 2019 through a third-party app collection website [68]. Apps are randomly collected from different time periods, with only one version of an app retained per period. As shown in Table IX, InconPreter identifies risky behaviors in 69 (26.44%) of the apps from 2010-2014 and 117 (15.64%) of the apps from 2015-2019. We calculate the percentage of apps containing each risky behavior across different periods. In 2010-2014, 22 risky behaviors appear in more than 10% of apps (11 risky behaviors appear in more than 40% of apps, with the highest percentage being 82.61%). In 2015-2019, 14 risky behaviors appear in more than 10% of apps (9 behaviors in more than 40% of apps, with the highest being 64.96%). In 2020-2024, 8 risky behaviors appear in more than 10% of apps (only one exceed 40%, with the highest being 41.16%). This indicates an improvement in the prevalence of risky behaviors in apps, possibly due to increasing attention from developers and users toward privacy and security concerns.

Additionally, we analyze the changes in the popularity rankings of risky behaviors across different periods. In all three periods, the top rankings remain almost unchanged, with **READ PHONE STATE**, **ACCESS NETWORK STATE**, and **WAKE LOCK** consistently appearing at the forefront. This could be attributed to these behaviors being fundamental and susceptible to misuse. Furthermore, we make some interesting observations. Risky behaviors related to user contact information (such as **READ CONTACTS**, **READ SMS**) have a declining trend in rankings, while those associated with Wi-Fi, Bluetooth, and other connectivity features (such as **ACCESS WIFI STATE**, **BLUETOOTH**) experience a rise in rankings. This trend could be attributed to the evolving nature of mobile phones, where interactions between users or devices increasingly involve Bluetooth, Wi-Fi, etc., rather than traditional communication methods like phone calls or text messages. Another interesting finding is that there's an increasing trend in the rankings of risky behaviors related to user location (such as **ACCESS FINE LOCATION**, **ACCESS COARSE LOCATION**). This could be attributed to the growing importance of location data for various services such as intelligent notifications and recommendations. Consequently, many apps may attempt to clandestinely collect such information. Although there are changes in the popularity rankings of risky behaviors, the frequency of these behaviors' occurrence has significantly decreased. Overall, mobile application privacy and security protection are moving towards a positive development trend.

V. Discussion

Deployment. InconPreter operates as an offline tool that analyzes app packages and provides detection results. App developers can locally deploy InconPreter, inputting the APK package to identify inconsistencies before app release or usage, to help provide users with a more reliable and satisfactory app service, leading to a better reputation and attracting more users. Third parties can deploy InconPreter for large-scale scanning, aiding in privacy governance within the mobile ecosystem.

For regular app users, similar to VirusTotal [67], InconPreter can be accessed through a web portal, allowing them to query by uploading app names/IDs or APKs. The analysis results not only help users avoid potential privacy breaches but also enable them to understand the actual behaviors of the app. InconPreter searches for inconsistent behaviors through the semantics of UI elements and code sequences, but not all inconsistent behaviors are necessarily risky. Some behaviors may be reasonable at the functional level. For example, when a user uses a gallery app to search for images, the app may access the network or directly read from a cache. These actions may not be apparent in the UI but are reasonable for such a task. InconPreter will present these app operations to users but not flag them as risky.

Insights. On the one hand, benign app developers and mobile system manufacturers can implement strategies to inform users in real time about the permissions an app requires and prompt users to confirm sensitive permissions when the app requests them. Currently, some systems, such as iOS and HyperOS, have implemented real-time monitoring of app permissions. These systems alert users when an app requests access to sensitive data or system resources. However, these notifications often lack detailed explanations regarding the specific reasons why the app requires such permissions, which can lead to user misunderstandings or even cause users to ignore these alerts altogether. Future improvements could involve implementing more informative and transparent permission request mechanisms. One effective approach could be to include comprehensible explanations alongside permission requests, specifying exactly how the requested permissions will be used and what benefits or potential risks are associated with granting them. Additionally, incorporating user-friendly design elements such as intuitive visual indicators, step-by-step walkthroughs, and examples of how the permissions will enhance the app's functionality could also improve user understanding and decision-making.

On the other hand, users should be cautious about granting sensitive permissions and remain vigilant about permissions that seem unrelated to the app's functionality. During app usage, users should pay attention to how their permissions are being utilized, particularly regarding the most common privacy breach of location access. Additionally, users can directly disable certain functionalities to prevent privacy misuse, such as turning off GPS location services. Our tool can also assist in identifying potential risks within apps, thus helping to avoid related issues.

Limitation. The limitation of this work mainly stems from the following two aspects.

Static Analysis. InconPreter utilizes static analysis to extract behaviors from app code. Consequently, it inherits common limitations of static analysis methods: it cannot guarantee that the extracted code will actually execute or that the UI information extracted statically will match what users see at runtime. The dynamic loading of interface elements and techniques like code reflection, where the app can modify its behavior or examine its structure at runtime, introduce complexities that static analysis tools may not account for. In future work, to further enhance the effectiveness of InconPreter, we can consider incorporating dynamic analysis methods, such as capturing fully loaded UIs during runtime to supplement and correct the statically extracted UI information. We can also design detection rules for common dynamic code loading techniques to improve the accuracy of call sequence extraction.

Additionally, FlowDroid is applied to extend the call graph, but its complex processing flow can lead to issues such as excessive memory usage and execution time. To enhance efficiency, we pruned FlowDroid's analysis. First, we focused on the data flow between Android system APIs and skipped analyzing irrelevant code from third-party libraries, such as ad libraries, thereby reducing the analysis scope. Second, since Android system APIs represent the smallest unit of behavior in our analysis, we terminate the analysis for a branch as soon as it reaches a system API, avoiding further exploration of its sub-calls. Pruning may affect the completeness of some data flows, but it saves resources. In the future, we can further explore and optimize the extent of the pruning.

Application of LLMs. InconPreter leverages LLMs to analyze risks among inconsistent behaviors. In this process, we employ the method of prompt engineering to acquaint the LLM with our background knowledge and accomplish the given tasks. However, due to the need for carefully designed prompts and continuous adjustment, the effectiveness may vary between different prompts, resulting in the inconvenience of human involvement. In the future, it may be worth considering the integration of prompt tuning methods to automatically explore prompts that yield satisfactory results. Additionally, the output results from a single large language model may exhibit some level of inaccuracy. We can consider using multiple large language models to determine the final results in further work.

Ethics. We conduct local experimental analysis of apps in a lab environment and report our findings through Google's feedback interface [69] in time. We will work collaboratively with their team on further verification and resolution efforts.

VI. Conclusions

The widespread adoption of mobile phones has resulted in a surge of apps, fulfilling user needs but also introducing various security concerns. In our research, we focus on extracting user-related behaviors from apps and explaining them to users in a comprehensible natural language form, empowering users to discern the disparities between users' real expectations and the app's actual behavior. We introduce InconPreter, which employs static analysis to initially extract app behaviors, followed by a series of filters to retain behaviors that may raise user concerns. Subsequently, inconsistent behaviors are identified by comparing the semantics of code operations with the app's UI content. Finally, LLM is utilized to interpret inconsistent behaviors in understandable natural language. Our evaluation demonstrates that InconPreter effectively extracts inconsistent behaviors from apps and provides accurate and reasonable explanations. Analyzing 10,878 wild apps obtained from Google Play, InconPreter discovers 1,664 risky inconsistent behaviors among 413 apps, highlighting significant threats to user data and device privacy and security. We conduct a thorough analysis of the discovered behaviors to deepen the understanding of inconsistent behaviors, thereby enhancing the privacy and security of the mobile ecosystem.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive comments. The IIE authors are supported in part by CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), NSFC (92270204, U24A20236) and Youth Innovation Promotion Association CAS.

References

[1] "Number of smartphone users worldwide from 2014 to 2029," https://www.statista.com/forecasts/1143723/smartphone-users-in-the-world, 2024.

[2] "Facebook has been collecting call history and sms data from android devices," https://www.theverge.com/2018/3/25/17160944/facebook-call-history-sms-data-collection-android, 2018.

[3] "Data protection act 2018," https://www.legislation.gov.uk/ukpga/2018/12/contents/ena cted, 2018.

[4] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds. ACM, 2010, pp. 73–84.

[5] A. P. Felt, K. Greenwood, and D. A. Wagner, "The effectiveness of application permissions," in *2nd USENIX Conference on Web Application Development, WebApps'11, Portland, Oregon, USA, June 15-16, 2011*, A. Fox, Ed. USENIX Association, 2011.

[6] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ser. SOUPS '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2335356.2335360

[7] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. A. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Symposium On Usable*

Privacy and Security, SOUPS '12, Washington, DC, USA - July 11 - 13, 2012, L. F. Cranor, Ed. ACM, 2012, p. 3.

[8] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 1025–1035.

[9] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, G. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 1354–1365.

[10] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 1036–1046. [Online]. Available: https://doi.org/10.1145/2568225.2568301

[11] C. Gatsou, A. Politis, and D. Zevgolis, "From icons perception to mobile interaction," in *Federated Conference on Computer Science and Information Systems, FedCSIS 2011, Szczecin, Poland, 18-21 September 2011, Proceedings*, M. Ganzha, L. A. Maciaszek, and M. Paprzycki, Eds., 2011, pp. 705–710.

[12] X. Xiao, X. Wang, Z. Cao, H. Wang, and P. Gao, "Iconintent: automatic identification of sensitive UI widgets based on icon classification for android apps," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 257–268.

[13] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu, and J. Lu, "Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 2421–2436.

[14] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "SUPOR: precise and scalable sensitive user input detection for android apps," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 977–992.

[15] T. T. Nguyen, D. C. Nguyen, M. Schilling, G. Wang, and M. Backes, "Measuring user perception for detecting unexpected access to sensitive resource in mobile apps," in *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, J. Cao, M. H. Au, Z. Lin, and M. Yung, Eds. ACM, 2021, pp. 578–592.

[16] J. Y. Khan and G. Uddin, "Automatic code documentation generation using gpt-3," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–6.

[17] "Codex," https://openai.com/blog/openai-codex/.

[18] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 426–436.

[19] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 627–638. [Online]. Available: https://doi.org/10.1145/2046707.2046779

[20] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 303–313.

[21] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, pp. 259–269.

[22] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, G. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 1329–1341.

[23] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 280–291.

[24] S. Arzt and E. Bodden, "Stubdroid: automatic inference of precise data-flow summaries for the android framework," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 725–735.

[25] K. Xu, Y. Li, and R. H. Deng, "Iccdetector: Icc-based malware detection on android," *IEEE Trans. Inf. Forensics Secur.*, vol. 11, no. 6, pp. 1252–1264, 2016.

[26] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society,

2017.

[27] X. Pan, Y. Cao, X. Du, B. He, G. Fang, R. Shao, and Y. Chen, "Flowcog: Context-aware semantics extraction and analysis of information flow leaks in android apps," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 1669–1685.

[28] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 94–105.

[29] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE Computer Society, 2017, pp. 23–26.

[30] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 245–256.

[31] A. I. Ali-Gombe, S. Sudhakaran, A. Case, and G. G. R. III, "Droidscraper: A tool for android in-memory object recovery and reconstruction," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 2019, pp. 547–559.

[32] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 291–307.

[33] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/intellidroid-targeted-input-generator-dynamic-analysis-android-malware.pdf

[34] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, T. A. Zia, A. Y. Zomaya, V. Varadharajan, and Z. M. Mao, Eds., vol. 127. Springer, 2013, pp. 86–103. [Online]. Available: https://doi.org/10.1007/978-3-319-04283-1_6

[35] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: effective and explainable detection of android malware in your pocket," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/drebin-effective-and-explainable-detection-android-malware-your-pocket

[36] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/dark-hazard-learning-based-large-scale-discovery-hidden-sensitive-operations-android-apps/

[37] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: eliminating experimental bias in malware classification across space and time," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 729–746.

[38] B. Wu, S. Chen, C. Gao, L. Fan, Y. Liu, W. Wen, and M. R. Lyu, "Why an android app is classified as malware? towards malware classification interpretation," *CoRR*, vol. abs/2004.11516, 2020. [Online]. Available: https://arxiv.org/abs/2004.11516

[39] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. M. Sadeh, S. M. Bellovin, and J. R. Reidenberg, "Automated analysis of privacy requirements for mobile apps," in *Network and Distributed System Security Symposium*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:655548

[40] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu, "Toward a framework for detecting privacy policy violations in android application code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 25–36.

[41] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," 07 2018, pp. 2269–2275.

[42] X. Yu, Q. Huang, Z. Wang, Y. Feng, and D. Zhao, "Towards context-aware code comment generation," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 3938–3947.

[43] Z. Yang, J. W. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 1–12, 2021.

[44] OpenAI, “Chatgpt [large language model],” https://openai.com/chatgpt/, 2024.
[45] ——, “Gpt-4,” https://openai.com/gpt-4.
[46] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, “Llama 2: Open foundation and fine-tuned chat models,” 2023.
[47] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020.
[48] T. Ahmed and P. Devanbu, “Few-shot training llms for project-specific code-summarization,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.
[49] E. Chen, R. Huang, H.-S. Chen, Y.-H. Tseng, and L.-Y. Li, “Gptutor: a chatgpt-powered programming tool for code explanation,” *arXiv preprint arXiv:2305.01863*, 2023.
[50] “Soot,” https://github.com/soot-oss/soot/.
[51] “Apktool,” https://apktool.org/.
[52] R. Smith, “An overview of the tesseract ocr engine,” in *ICDAR 2007*, vol. 2. IEEE, 2007, pp. 629–633.
[53] “Acronym finder,” https://www.acronymfinder.com/.
[54] “Abbreviations.com,” https://www.abbreviations.com/.
[55] “Android open source project (application security),” https://source.android.com/security/overview/app-security, 2022.
[56] “Android api reference,” https://developer.android.com/reference.
[57] “spiral,” https://github.com/casics/spiral/.
[58] “Nltk,” https://www.nltk.org/.
[59] A. Rahali, A. H. Lashkari, G. Kaur, L. Taheri, F. Gagnon, and F. Massicotte, “Didroid: Android malware classification and characterization using deep image learning,” in *ICCNS 2020: The 10th International Conference on Communication and Network Security, Tokyo, Japan, November 27-29, 2020.* ACM, 2020, pp. 70–82. [Online]. Available: https://doi.org/10.1145/3442520.3442522
[60] H. C. Wu, R. W. P. Luk, K. Wong, and K. Kwok, “Interpreting TF-IDF term weights as making relevance decisions,” *ACM Trans. Inf. Syst.*, vol. 26, no. 3, pp. 13:1–13:37, 2008. [Online]. Available: http://doi.acm.org/10.1145/1361684.1361686
[61] “spacy,” https://spacy.io/.
[62] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, “Pscout: analyzing the android permission specification,” in *the ACM Conference on Computer and Communications Security, CCS’12, Raleigh, NC, USA, October 16-18, 2012*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM, 2012, pp. 217–228. [Online]. Available: https://doi.org/10.1145/2382196.2382222
[63] R. W. McGee, “Annie chan: Three short stories written with chat gpt,” *Available at SSRN 4359403*, 2023.
[64] R. J., “Chatgpt and legal writing: the perfect union?” *SSRN*, 2023.
[65] T. E. Cox C, “Chatgpt: implications for academic libraries,” *College Research Libraries News*, p. 84(3): 99, 2023.
[66] “Android malware dataset,” http://amd.arguslab.org/.
[67] “Virustotal,” https://www.virustotal.com/.
[68] “Apkcombo,” https://apkcombo.com/, 2018.
[69] “Report a policy violation,” https://support.google.com/googleplay/android-developer/contact/policy_violation_report.

VII. Appendix

A. Study Questionnaire

In the first questionnaire, we presented participants with 7 UI samples from different app functionalities, along with the analysis results like that shown in Fig. 2d. For each sample, we ask participants the following questions:

1) How easy is it to understand the provided explanation? Please rate it from 1 to 5, with 5 being the easiest to understand.
2) How reasonable do you think the provided explanation is? Please rate it from 1 to 5, with 5 being the most reasonable.
3) Were you previously aware that mobile apps might exhibit similar behaviors?
4) How concerned are you about the potential impact of this behavior on your data privacy or device security? Please rate it from 1 to 5, with 5 being the most concerned.
5) How helpful do you think our explanation is in enhancing your awareness of potential risks? Please rate it from 1 to 5, with 5 being the most helpful.

In the second questionnaire, we show participants all detected risky inconsistent behaviors and provide explanations and potential risks for these behaviors. For each behavior, we ask participants the following two questions:

1) How well do you understand what this behavior is doing? Please rate it from 1 to 5, with 5 being the most understanding.
2) Some apps may engage in this behavior without your awareness, and this behavior is not necessary for the app to perform its claimed functionality. How concerned are you about the potential impact of this behavior on your data privacy or device security? Please rate it from 1 to 5, with 5 being the most concerned.

B. Risky inconsistent behaviors

TABLE X: Risky inconsistent behaviors extracted

Behavior	**Description**
READ PHONE STATE	read device status and identity information, including the device's phone number, current cellular network information, any ongoing calls, and the device's IMEI/ESN, posing potential risks of privacy leakage and identity theft.
ACCESS NETWORK STATE	access information about the device's network connection status, such as the connection to Wi-Fi or mobile data networks, revealing the user's network activity information.
WAKE LOCK	prevent the device from entering sleep mode to ensure it stays awake during specific operations, which accelerates battery and resource consumption.
ACCESS COARSE LOCATION	access coarse device location information, such as location based on mobile networks or Wi-Fi, resulting in user location privacy leakage and potential location tracking.
ACCESS FINE LOCATION	access precise location information from the device, including GPS location, Wi-Fi location, Bluetooth location, etc.
WRITE EXTERNAL STORAGE	save data to public directories like the SD card and application-specific directories, including creating, modifying, and deleting files.
ACCESS WIFI STATE	access the Wi-Fi connection status information of the device, including details about the current Wi-Fi network connection and the list of available Wi-Fi networks nearby, may result in tracking the user's network activities.
INTERNET	send and receive data over the internet, which may lead to issues such as sensitive data transmission and mobile data usage.
SEND SMS	send unauthorized SMS messages.
VIBRATE	control the vibration function, causing unnecessary disturbance to users.
READ EXTERNAL STORAGE	read from the device's storage space, involving accessing any files stored on the device, such as images, audio, video, and documents.
BLUETOOTH	connect to Bluetooth devices and communicate with them.
READ CONTACTS	access the list of user contacts on the device, including names, phone numbers, email addresses, and other information.
WRITE SETTINGS	modify system settings, including but not limited to sound, display, input method, date and time settings.
GET ACCOUNTS	access account information stored on the device, including local accounts and those synced with the device, such as social media accounts and email accounts.
SYSTEM ALERT WINDOW	display overlay content on the user interface, such as floating notifications, menus, or tools, may disrupt the user's normal interaction with other applications, or entice them to click on malicious links or perform risky actions.
CHANGE WIFI STATE	modify the Wi-Fi connection status, such as enabling or disabling the Wi-Fi feature and connecting to specific Wi-Fi networks, disrupting the network connection.
READ SMS	read SMS messages.
CAMERA	access the device's camera, allowing the ability to take photos or record videos.
USE ICC AUTH WITH DEVICE IDENTIFIER	communicate with the SIM card and authenticate with the device identifier, allowing apps to interact with carrier services, such as sending messages and receiving calls.
BROADCAST STICKY	send sticky broadcasts, which remain in the system after being sent, allowing them to be received later or by unregistered receivers, which may lead to the dissemination of sensitive information and excessive use of device resources.
MODIFY AUDIO SETTINGS	modify audio settings, including volume, channels, etc.
RECEIVE SMS	receive SMS messages.
CALL PHONE	initiate unauthorized phone calls.
RECORD AUDIO	record audio.
SCHEDULE EXACT ALARM	precisely schedule timers to trigger operations or events at specified time.
WRITE CONTACTS	modify, add, or delete user contacts on the device.
INSTALL PACKAGES	install other apps.
NEARBY WIFI DEVICES	scan nearby Wi-Fi devices, which may lead to tracking and location identification.
CHANGE NETWORK STATE	modify device network connection status, including enabling or disabling network connections, changing network connection types, etc., may result in unstable network connections, increased mobile data usage, network hijacking, and other issues.
BLUETOOTH SCAN	scan nearby Bluetooth devices, which may lead to tracking and location identification.
FOREGROUND SERVICE	display persistent notifications in the notification bar, which may disrupt the user's usage experience, and impact device performance and battery life.