2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-1594-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISSRE59848.2023.00057

APIMind: API-driven Assessment of Runtime Description-to-permission Fidelity in Android Apps

Shaokun Zhang, Hanwen Lei, Yuanpeng Wang, Ding Li†, Yao Guo†, Xiangqun Chen
Key Lab of High-Confidence Software Tech (MOE), School of Computer Science, Peking University, Beijing, China
{skzhang, yuanpeng_wang, ding_li, yaoguo, cherry}@pku.edu.cn, lei_hanwen@stu.pku.edu.cn

***Abstract*—Assessing description-to-permission fidelity is critical for safeguarding personal data accessed through sensitive APIs in Android apps. However, it remains a challenge for existing methods, both static and dynamic. Static methods are either infeasible due to various dynamic features (e.g., code obfuscation, dynamic class loading, and reflection) or too coarse-grained to understand how sensitive APIs collect privacy data under runtime contexts. Existing dynamic methods lack contextual understanding regarding sensitive API calls. For example, they fail to understand which GUI widgets are more likely to trigger sensitive APIs and ignore the preceding UI contexts that could reveal the intention of API calls when analyzing their fidelity.**

In this paper, we propose an API-driven automated dynamic analysis tool called *APIMind* for assessing runtime description-to-permission fidelity in Android apps. *APIMind* can discover sensitive APIs more effectively by utilizing multimodal features to jointly infer the semantics of GUI widgets and leveraging deep networks to automatically learn their relationship based on multi-faceted rewards. Then, it could accurately assess description-to-permission fidelity by developing an extended tool that considers dual UI contexts (i.e., preceding and current contexts). We evaluate the accuracy and efficiency of *APIMind* using 121 real-world apps. Experimental results demonstrate that *APIMind* can achieve a detection accuracy of 96.1%. Compared to the competitive baseline, *APIMind* increases efficiency by 43%. In addition, based on our proposed tool, we conduct a large-scale case study of 1013 real Android apps, which reveals the prevalence of several typical inconsistencies and demonstrates the effectiveness of our approach in the wild.

I. Introduction

Mobile devices have become an indispensable part of modern daily life, with those running Android accounting for around 84% of the global smartphone market as of 2022 [1]. People often use mobile apps for various purposes, such as entertainment, shopping, and learning. However, these apps may also gather and hold a significant amount of sensitive data, such as contacts, photos, and locations. To ensure user privacy, the Android system mandates that mobile apps request permission before accessing sensitive data. Nevertheless, the question remains: are these permissions truly consistent with the app's functionalities and descriptions? If an app requests unnecessary or excessive permissions, it may use personal data improperly, resulting in potential risks and losses. Therefore, assessing description-to-permission fidelity[1] is crucial for protecting personal data in Android apps [2]–[4].

Previous efforts to analyze description-to-permission fidelity fall into two categories: static approaches that perform a comprehensive source-level review of the APK file [3], [5]–[9] and dynamic approaches that instrument and explore the app in the runtime context [2], [10]–[15]. For static approaches, a straightforward way is to correlate app descriptions with requested permissions, such as AUTOCOG [3]. However, they are too coarse-grained to understand how their sensitive data is collected under runtime contexts. Some studies, such as FLOWCOG [16], attempt to perform flow-level operations with existing static analysis tools [6] to analyze the semantics of sensitive information flows. Nevertheless, static analyses for Android apps are often infeasible [2], [17], [18] due to the intense use of dynamic features such as code obfuscation, code encryption, dynamic class loading, and reflection (especially in malware apps).

Dynamic analysis techniques mainly include two phases: one is to trigger sensitive API calls effectively, and the other is to analyze the legitimacy of sensitive API calls in a given UI context. APICOG [2] is a representative that leverages a general testing framework to discover sensitive APIs. They attempt to correlate a sensitive API call with its UI state and extract semantics from them using natural language processing (NLP) models. Based on the extracted semantics, machine learning models can be used to justify the legitimacy of the calls. However, they lack contextual understanding regarding sensitive API calls, which involves two main aspects.

On the one hand, they rely on a general testing framework, which is not optimized for exploring sensitive API calls. In other words, they cannot understand which GUI widget is more prone to trigger sensitive API calls. This could result in wasted time and missed detections of sensitive APIs. On the other hand, when determining the legitimacy of a call, only the current UI context is considered, which limits the detection capability. They ignore the preceding UI context, such as the executed widget and its context (e.g., user agreements), which may describe the purpose or intent behind a sensitive API call.

To address these issues, we propose a novel automated dynamic analysis tool called *APIMind* to assess runtime description-to-permission fidelity in Android apps. *APIMind* consists of two major components: Trigger and Fidelity Analyzer. Specifically, to address the first issue, we propose an API-driven automated tool, Trigger, to thoroughly explore and prioritize activities based on their likelihood of accessing sensitive APIs. To address the second issue, we propose

†Corresponding authors.

[1]Fidelity issues occur when sensitive APIs are not used appropriately in the activities that involve their context.

2332-6549/23/$31.00 ©2023 IEEE
DOI 10.1109/ISSRE59848.2023.00057

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

an extension of APICOG, a dual-context Fidelity Analyzer that examines the legitimacy of a sensitive API call, which considers both current and preceding UI contexts.

Realizing *APIMind* mainly involves three challenges, the first of which is the difficulty of understanding GUI widgets. One intuitive way is to analyze their visual appearance. However, it is challenging to extract discriminative and robust features of GUI widgets because there are many uncertainties and noises in GUI images, such as widget state changes (e.g., a "next" button turns from unavailable to available when data collection is complete.), and visually similar GUI widgets. Another possible way is to understand their semantics through the corresponding meta-attributes. However, many GUI widgets lack any text or content descriptions indicating their purpose [19]. The second challenge is that the relationship between semantic features of GUI widgets and sensitive API calls is unclear.

To address the above challenges, we develop a deep learning model to jointly infer the semantics of GUI widgets by leveraging multimodal features, including fine-grained meta-attributes, visual features, and layout context, which enables a more comprehensive understanding. Moreover, this model is designed to explicitly maximize multi-faceted rewards, which include the API-level reward that prioritizes activities with more sensitive API calls. This allows the model to learn meaningful relationships between GUI widgets and sensitive APIs. The third one is that there are not enough labeled training samples. Training deep learning models typically requires a large number of samples, which are scarce in this scenario. We use a reinforcement learning-based model that autonomously generates training data and integrates it into the learning algorithm. This data is generated by simulating user interactions with GUIs, creating a more diverse set of training examples, and improving the robustness of the model.

We evaluate the performance and efficiency of *APIMind* using 121 realistic apps from the Xiaomi market. The results demonstrate that *APIMind* can achieve a detection accuracy of 96.1%. Regarding efficiency, *APIMind* exhibits a 43% improvement compared to the competitive baseline. Based on our newly proposed tool, we conduct a large-scale case study involving 1013 real Android apps. Our analysis reveals the prevalence of several typical inconsistencies: unnecessary permissions, deceptive agreements, and retaining permissions. In addition, 82.8% of apps contain at least one inconsistency, indicating a significant prevalence in Android apps. Furthermore, we have also reported some of the detected inconsistencies to the platform administrator, in order to demonstrate the effectiveness of our approach.

The contributions of this paper are summarized as follows.

- We propose a novel automated tool called *APIMind* to assess runtime description-to-permission fidelity in Android apps. It can trigger sensitive APIs more effectively by utilizing multimodal features, while leveraging deep networks to automatically learn their relationship based on multi-faceted rewards.
- We conduct a large-scale case study on runtime description-to-permission fidelity in Android apps. Our findings reveal the prevalence of several typical inconsistencies, which could offer valuable insights into developing more effective solutions.
- We extensively evaluate our approach using a diverse range of Android apps gathered from the Xiaomi App Store. The experimental results demonstrate the accuracy and effectiveness of *APIMind* in assessing runtime description-to-permission fidelity.

Data Availability: In addition, our tool is publicly available on GitHub (https://github.com/skzhangPKU/APIMind).

II. Background and Motivation

In this section, we present the background and motivation of this work. First, we provide the preliminaries of Android GUI programming and Deep Reinforcement Learning. Then, we introduce the limitations of current approaches in our task, which involves two subtasks: Sensitive API Triggering and Fidelity Assessment.

A. Background

1) Android GUI Programming: Activities constitute fundamental components of apps in Android development. Ordinarily, it occupies the entire display, although smaller ones are applicable for multi-window mode or floating windows. Widgets are small GUI components that enable users to interact with apps, such as buttons, text fields, and toggles. A frame GUI window provides a top-level container for various activities, providing a structural basis for their rendering and interaction. Through an Intent object, an activity can be started by another, with the system controlling the transitions between them. A GUI transition graph can represent an app's functionality and behavior through a directed graph, defined as $G = (V, E)$ where V denotes the nodes representing individual activities within the app, and E denotes the edges representing the available transitions between these nodes.

2) Deep Reinforcement Learning: Deep Reinforcement Learning (DRL) aims to guide agents to make decisions based on feedback from the environment by integrating deep learning and reinforcement learning techniques. Its objective is to learn an optimal policy, i.e., a mapping from states to actions, that maximizes the expected cumulative reward. One of the most well-studied DRL algorithms is the Deep Q-Network (DQN), proposed by Mnih *et al.* [20]. The network receives the current state of the environment as input, yielding an output Q-value for each possible action. The agent selects its action by choosing the one with the highest Q-value at each step. To train the network, DQN employs a replay memory mechanism to store experience tuples of the form (state, action, reward, next_state). These tuples are randomly sampled during training to reduce the correlation between successive updates.

B. Motivation

1) Sensitive API Triggering: To efficiently execute widgets that invoke sensitive API calls, a straightforward way is to use

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

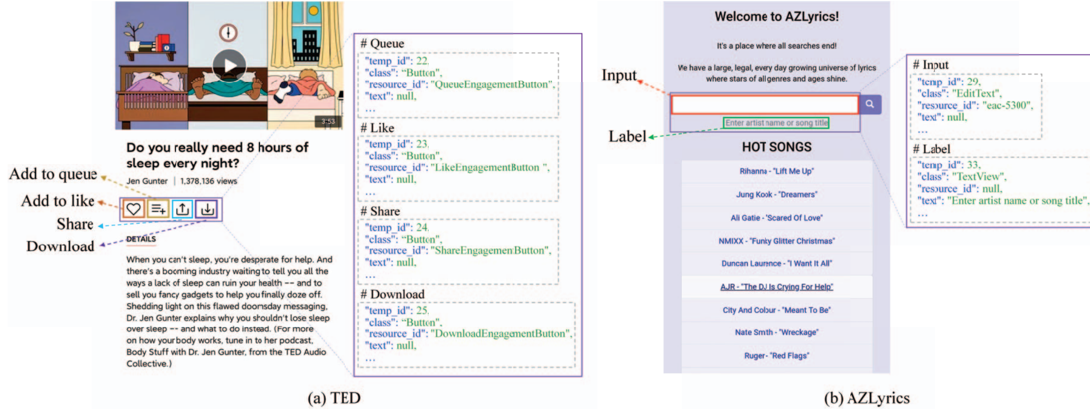

Fig. 1: A motivating example based on the TED and AZLyrics apps. The TED app provides talks from remarkable researchers on various topics. The AZLyrics app is one of the biggest databases of song lyrics worldwide. Specifically, Figure (a) shows a detail page for a particular talk, while Figure (b) shows a page with the function to search for lyrics by providing song titles or artist names.

Fig. 2: An illustrative example of a reading app.

a GUI tester [21] that randomly produces GUI operations [22] or relies on heuristics [23], [24] to generate them. Unfortunately, these testers are not designed to efficiently identify sensitive API calls. Therefore, there is an urgent need for a new testing approach that goes beyond traditional GUI testers. To achieve this goal, understanding the semantics of GUI widgets is necessary, which can enable automated tools to generate more targeted and intelligent GUI operations that reveal sensitive API calls. However, it is particularly challenging due to the uncertainty of GUI widgets (e.g., the button state changes from enabled to disabled) and visually similar GUI widgets (e.g., input boxes look identical for all GUI states). Besides, for some widgets, even humans have difficulty understanding them based on vision alone. For instance, in Fig. 1(a), the meaning representation of the GUI widgets marked by the orange and light blue boxes is potentially confusing. For current approaches, like [25]–[32], most tend to comprehend GUI states as a whole, which is too coarse-grained to infer the semantics of local widgets.

2) Fidelity Assessment: For fidelity assessment, APICOG [2] is a representative that checks the legitimacy of sensitive API calls by correlating them with their current UI contexts. However, they ignore the preceding UI context, which may describe the purpose or intention behind a sensitive API call. Fig. 2 presents an illustrative example. In this example, the preceding UI context refers to the executed widget marked by the red box and its adjacent widgets (i.e., the user agreement) in Fig. 2(a). The current UI context refers to the context shown in Fig. 2(b). It shows a prompt box describing a reading app requesting device identifiers for account security control and statistics, as shown in Fig. 2(a). Upon agreeing to the authorization, the user will click the button marked by the red box in Fig. 2(a), which leads to the main page displayed in Fig. 2(b). The main page displays a ranking list of e-books, yet device identifiers are not essential for this page. Consequently, the APICOG concludes that there is an inconsistency between the sensitive data collected and user-expected behaviors. Indeed, however, such access to sensitive data is compliant because the preceding UI context claims its intention for legitimate purposes.

III. Design of *APIMind*

To automatically assess runtime description-to-permission fidelity, we present *APIMind*, a novel automated tool that detects inconsistencies between contexts and the collected privacy data in Android apps. Fig. 3 illustrates the workflow of *APIMind*. *APIMind* receives a given APK file as input and delivers a detailed execution report that highlights any detected inconsistencies. It comprises two main components: Trigger, which exhaustively examines activities within the target app and monitors all attempts to access sensitive APIs, and Fidelity Analyzer, which detects any possible inconsistencies between the user-expected behavior and access to sensitive APIs.

A. Trigger

Trigger examines an Android app by conducting a comprehensive exploration of its activities. Throughout the examination, Trigger generates the screenshots, layout files, and permissions of the activities and sends them to the Fidelity Analyzer to identify potential inconsistencies. The Trigger architecture is shown in Fig. 4.

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

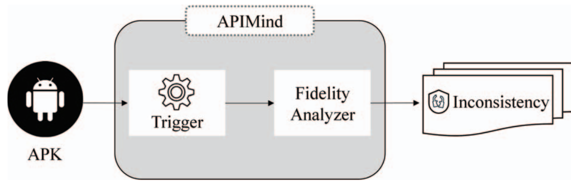

Fig. 3: System overview of *APIMind*.

The goal of Trigger is to efficiently execute widgets that invoke sensitive API calls. Unlike previous methods [26], [30], [33], Trigger considers not only visual images perceivable by a human tester but also fine-grained meta-attribute features of GUI widgets and layout contexts. Meta-attribute features can reveal the behavior, appearance, and functionality widgets provide [34]. For example, in Fig. 1(a), the semantics of GUI widgets marked with different colored boxes can be inferred from the texts extracted from the "resource_id" attributes, such as "Queue", "Like", "Share", and "Download". However, previous studies [19], [35] reported that there are still many GUI widgets with missing or incorrect attributes due to the poor development behaviors of programmers. As shown in Fig. 1(b), the "text" attribute value of the input box is missing, and its "resource_id" attribute value is "eac-5300" without any practical implications. This suggests that there is a need to infer the semantics based on its context, such as surrounding descriptive texts. In this example, the corresponding text is "Enter artist name or song title". Based on the above analysis, we jointly infer the semantics of GUI widgets using multi-modal features that simultaneously consider fine-grained meta-attribute features, visual features, and layout context features.

Another challenge is that the relationship between the semantic features of GUI widgets in activities and sensitive APIs is unclear in runtime contexts. To address this challenge, we leverage deep models to automatically extract such relationships from contexts to understand which GUI widgets are more likely to trigger sensitive APIs. However, modeling the complex relationship usually requires a large number of samples. In the case of triggering sensitive APIs, it is difficult to obtain such quantities of samples. To address this issue, we utilize a reinforcement learning-based model that autonomously generates training data and integrates it into the learning algorithm. To this end, we model Trigger as a Reinforcement Learning (RL) agent with a deep learning model that automatically prioritizes activities based on their likelihood of invoking sensitive APIs. Thus, Trigger can focus on activities related to sensitive APIs.

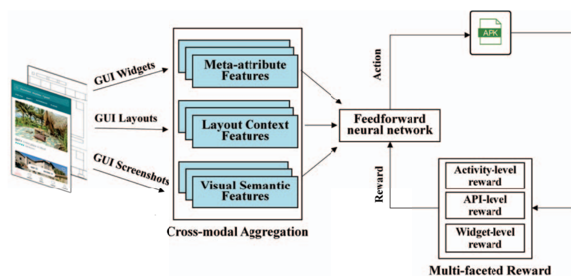

Fig. 4: The Trigger process.

Specifically, Trigger is an RL agent based on DQN [36] that navigates through the transition graph of activities in an Android app. Its main component is the Q-Network, which predicts the reward for each available action based on the current state. The Q-Network is implemented as a feedforward neural network, which can effectively model the intricate dependencies among activities and sensitive APIs, owing to its ability to approximate complex functions [37]. Next, we will introduce the core components of Trigger, including cross-modal aggregation, multi-faceted reward, and its training details.

1) Cross-modal Aggregation: The architecture of the Q-Network comprises three dense layers, as depicted in Equation 1.

$$f(X) = F_3(\sigma(F_2(\sigma(F_1(X;\Theta_1));\Theta_2));\Theta_3) \tag{1}$$

where Θ_* is learnable weights, F_* denotes the linear transformation performed by the dense layer, σ is the activation function, and X is a compressed encoding of the state, represented as a cross-modal aggregation of distinct features. Specifically, the Q-network incorporates three types of inputs for an activity, including meta-attribute features, visual semantic features, and layout context features. These features are fused to obtain a state representation using a concat-attention layer [38].

Meta-attribute features. A GUI widget includes many meta-attributes, such as texts and placeholders. The placeholder attribute refers to a short prompt describing the field's expected value (e.g., a brief description of the desired format or a sample value). These attributes can be easily extracted from dumped layout files. We feed the attributes into a pre-trained multilingual sentence embedding model, Sentence-Transformer [39], which produces a 768-dimensional feature vector.

Visual semantic features. GUI states of activities are the most sensory observation a human tester perceives from a mobile device. To obtain visual features, we send the GUI screenshot to AugNet [40], an unsupervised visual representation network that generates a 768-dimensional feature vector. The AugNet model is trained on a large-scale public mobile GUI dataset RICO [41] containing over 66k unique samples. Although this model is trained from different apps, it is beneficial for understanding activities and widgets because extracted visual features are considered globally distinguishable [42].

Layout context features. As mentioned earlier, the layout context helps to infer the semantics of GUI widgets. Following previous research [43], we use a self-supervised embedding model LayoutAutoEncoder [44] to extract layout context features from the hierarchy of activities, resulting in a numerical vector with 64 dimensions. Specifically, it comprises a two-step process. The first step is to assign each pixel of a GUI screenshot to a different category based on the characteristics of the widget to which it belongs. The second is to encode the

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

layout as a bitmap and extract its structural semantics using autoencoders.

2) Multi-faceted Reward: The Q-Network aims to forecast the likelihood of upcoming activities invoking sensitive APIs. This can be achieved by accomplishing two sub-objectives: (1) exploring more activities that involve sensitive APIs and (2) discovering additional new activities. To this end, a multi-faceted reward system is designed to jointly address these two sub-objectives. The multi-faceted reward comprises three segments, i.e., the API-level reward, the activity-level reward, and the widget-level reward. Specifically, the API-level reward satisfies the first sub-objective by prioritizing activities with a greater frequency of sensitive API requests. On the other hand, the remaining two rewards concentrate on accomplishing the second sub-objective by discovering additional new activities. Formally, the multi-faceted reward is defined as:

$$R_i = \alpha R_i^p + \beta R_i^a + \gamma R_i^w \tag{2}$$

where R_i^p, R_i^a, and R_i^w denote the API-level reward, activity-level reward, and widget-level reward, respectively, and α, β, and γ are weighting factors that determine the respective significance of individual items.

We define the **API-level reward** R_i^p as the count of unique sensitive APIs invoked by an activity. The **activity-level reward** is defined by Equation 3, which enables the Trigger to prioritize activities that exhibit significant differences. Specifically, the variable definitions in Equation 3 are as follows: X represents the current activity screenshot, Y corresponds to the screenshot of the succeeding activity in the GUI transition graph, τ is the distance threshold, EB is the embedded model AugNet [40] that transforms screenshots to numerical vectors, and $dist(\cdot,\cdot)$ is the Euclidean metric.

$$R_i^a = \begin{cases} 1, & \text{if } dist(EB(X), EB(Y)) > \tau \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

The **widget-level reward** is defined as Equation 4, which balances two preferences: visiting activities with more unvisited widgets and operating widgets with lower visit frequency.

$$R_i^w = M \times (\frac{UN_{sa}}{N} + \frac{1}{VF_{ow}}) \tag{4}$$

where VF_{ow} denotes the visiting frequency of operated widgets, UN_{sa} represents the number of unvisited widgets in the succeeding activity, and M and N are scaling factors to ensure their comparability

The underlying idea behind the widget-level reward is that transitions between activities are frequently initiated by widget operations, such as a button click. Consequently, an activity with more unvisited widgets increases the likelihood of transiting to a new one. Therefore, we integrate the widget-level reward with the activity-level reward to effectively guide the Trigger to new activities.

3) Trigger Learning: Algorithm 1 summarizes the learning process of the Trigger. The details are as follows. When training an app, Trigger first checks the existence of the behavior model. If no such model exists, a new model is created (Line 2). The replay buffer is initialized to store past experiences (Line 3). Then, Trigger launches the app under test (AUT) to access the first UI page (Line 4). Taking the page's GUI screenshot and layout file as input, the algorithm extracts multi-modal features, including visual features, layout context features, and fine-grained meta-attribute features, from them as the initial state (Line 5).

Algorithm 1: The *APIMind* Learning Process.

```
   Input: AUT; DQN model M; Time budget T.
   Output: Trained behavior model M;
 1 if M not exist then
 2   M ← initializeNewModel();
 3 Initialize replay buffer 𝒟 ← ∅;
 4 u_0 ← launch(AUT);
 5 s_0 ← fuseMultiModal(u_0);
 6 for t = 0,T do
 7   a_t ← selectAction(s_t,M);
 8   u_{t+1}, widgetStats, apiTrig ← execute(AUT, a_t);
 9   r_t^p ← calcAPIReward(apiTrig);
10   r_t^a ← calcActivityReward(u_t,u_{t+1});
11   r_t^w ← calcWidgetReward(widgetStats);
12   r_t ← calcImmediateReward(r_t^p, r_t^a, r_t^w);
13   s_{t+1} ← fuseMultiModal(u_{t+1});
14   𝒟 ← 𝒟∪ (s_t, a_t, r_t, s_{t+1});
15   B ← randomSampleBatch(𝒟);
16   M ← updateModel(B,M);
17   s_{t+1} ← s_t;
```

Next, the iterative processing continues until the time budget is exhausted (Lines 6-17). For each iteration, the algorithm leverages deep models to predict the reward values of all potential actions and selects and executes the one a_t with the highest reward (Lines 7-8). After the action execution step, the resulting information, containing the next UI page, triggered sensitive API calls, and widget statistics, are used to calculate the reward r_t (Lines 9-12). With the page, the algorithm converts it to a new state s_{t+1} (Line 13). The above process produces a 4-tuple (s_t, a_t, r_t, s_{t+1}), which is put into the buffer. To train the deep model, it randomly samples a mini-batch from the buffer and updates the model on the batch using temporal difference learning [45] (Lines 15-17).

B. Fidelity Analyzer

Fidelity Analyzer assesses whether an activity gathers sensitive data beyond what is necessary. It takes a pair $< activity, API >$ as input while outputting a binary label indicating whether the semantics of the API align with the contexts.

We propose an extended APICOG to determine whether the context is consistent with its collected privacy data. The high-level idea of the original APICOG [2] is as follows. First, it fetches the text in the screenshot, text-typed attributes of GUI widgets, app descriptions, and descriptions of the called sensitive APIs. It then cleans the raw text and converts them

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

TABLE I: The list of monitored sensitive APIs

Resource Type	API Class	Monitored API Example	Description
Telephony	TelephonyManager, SmsManager	getLine1Number(), getDeviceId(), sendTextMessage()	Retrieve sensitive information related to telephony
Location	Location, Address, GoogleMap, LocationManagerService	getLastKnownLocation(), requestLocationUpdates(), getLongitude()	Access user's location information, including GPS coordinates
Bluetooth	BluetoothDevice, BluetoothAdapter	getName(), getAddress(), getBondedDevices()	Potentially track user movement or identify nearby devices
Microphone and Camera	AudioRecord, MediaRecorder, Camera	startRecording(), start(), takePicture()	Access microphone and camera to record audio and video
Connectivity States	WifiManager, WifiServiceImpl	getScanResults(), getDhcpInfo(), getWifiState()	Retrieve information about user's network connectivity and usage patterns
Application States	PackageManager, UsageStatsManager	getInstalledPackages(), getApplicationInfo(), queryUsageStats()	Examine the accessibility of installed antivirus and financial apps to detect attacks such as phishing
Accounts	AccountManager	getAccountsByType()	Access and manage user's account credentials on the device
Contents	ContentProvider	query()	Manage app data storage

to a group of valid verb-noun pairs using Stanford Parser [46]. Finally, taking these pairs as input, it utilizes a machine learning model to determine whether they are consistent. However, as mentioned earlier in Section II-B, the original APICOG only considers the current UI context and ignores the preceding context, which limits its detection capabilities.

To address this issue, we propose a dual-context Fidelity Analyzer which considers preceding and current UI contexts when determining the legitimacy of a sensitive API call. We incorporate the preceding context to reduce misjudgments, as it could reveal the purpose or intention of the sensitive API call, such as account security control and statistics in Fig. 2(a). Notably, We use only one preceding activity as the context because it captures the most recent and relevant information while reducing noise and redundancy from multiple preceding activities. The preceding context is incorporated into the model, similar to the current context integrated into the original APICOG. Apart from the preceding context, we are consistent with the original APICOG in all other aspects. In other words, the preceding context is also extracted and converted into a valid set of verb-noun pairs, which are then fed into the machine learning model along with other pairs for fidelity assessment. Due to the page limit, we omit the details of Fidelity Analyzer, which can be found in the original APICOG paper [2].

IV. Evaluation

In this section, we perform extensive experiments to evaluate the proposed tool. The evaluation aims to answer the following research questions.

- **RQ1: How effective is *APIMind* in discovering sensitive API calls?**
- **RQ2: How effective is *APIMind* in assessing description-to-permission fidelity in Android apps?**
- **RQ3: How do multimodal features affect the performance of *APIMind*?**
- **RQ4: How prevalent are typical inconsistencies among apps?**
- **RQ5: Are the detected inconsistencies helpful to the administrators?**

A. Experiment Setup

Our experiments are conducted on an Ubuntu server version 18.04 LTS, equipped with a CUDA-enabled Nvidia GTX 1080 Ti GPU featuring 11GB memory. All apps are executed on a Google Pixel 5, a real mobile device with root access running on Android. We deliberately avoid using an emulator because if malware detects itself in an emulator environment, it will block sensitive data leakage [47].

APIMind automates apps through Uiautomator2 [48]. It is developed and implemented using the Pytorch framework [49]. *APIMind* utilizes Frida [50] to instrument apps and dynamically monitor sensitive API calls. To achieve this, we develop JavaScript snippets to interact with the Frida server by injecting them into apps.

Based on the previous research [16], we limit the exploration of each app to a maximum of 20 minutes. Moreover, the experimental parameters are set as follows. The weighting factors for the API-level, activity-level, and widget-level rewards are set at 50, 2, and 1, respectively. We set scaling factors for widget-level rewards as $M = 2$ and $N = 10$. We set the distance threshold τ to 2. A small-scale experiment is conducted to establish best practices, upon which all hyper-parameters for *APIMind* are determined. Moreover, we recruit three senior software engineering undergraduates familiar with Android app development to label the samples by majority voting.

We employ a systematic approach to construct the dataset for the experiments. Our first step is to develop a crawler that could get hold of apps from the Xiaomi app store(https://app.mi.com/download/{id}). Our crawler generates app links with "id" values between 1 and 1000. After eliminating the invalid links, the resulting dataset consist of

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

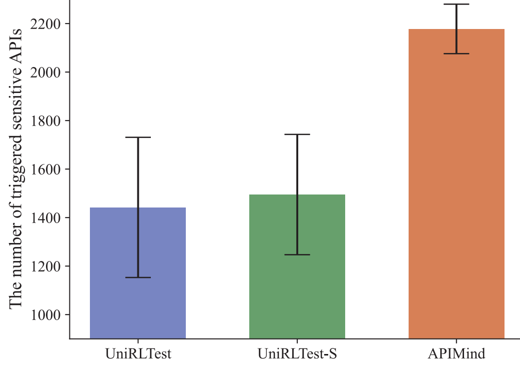

Fig. 5: Quantitative results of *APIMind*, *UniRLTest-S*, and *UniRLTest* in discovering sensitive API calls.

605 apps. We split these into training and testing sets of 484 and 121 apps, respectively. These apps generally vary in size, ranging from 128KB to 424 MB. Furthermore, based on previous research [2], [51], [52], we identify a set of sensitive APIs to be monitored during our experiments, shown in Table I.

B. RQ1: Effectiveness in discovering sensitive APIs

To demonstrate the effectiveness of our tool in identifying sensitive APIs, we design a set of experiments to undertake a thorough evaluation. Specifically, we measure the speed of invoking sensitive APIs by *APIMind* and compare it against two baseline methods. The first one is a state-of-the-art RL GUI testing framework (*UniRLTest*) [53] that focuses on maximizing coverage. It is more recent and efficient than Droidbot [54], which is the GUI tester used by APICOG. The second one (*UniRLTest-S*) is a modified version of *UniRLTest* that incorporates the number of sensitive API calls in each activity into its objective function. This method examines the validity of our Q-Network design and multi-faceted reward scheme.

We implement uniform training settings for all the methods. Specifically, we follow the precedent of previous researchers [16] and conduct training and exploration on each app for a maximum of 20 minutes. The exploration process is repeated three times to guarantee reliable results because various training experiences could prompt the model to pursue different strategies [26]. Furthermore, descriptive statistics, including the mean, standard deviation, and median, are performed on sensitive API calls identified by the three methods. By analyzing these metrics, we could compare the effectiveness of these methods.

Fig. 5 provides the quantitative results of the proposed approach and other baselines. It can be clearly found that *UniRLTest* and *UniRLTest-S* rarely detect sensitive API calls. The reasons for this observation are as follows. *UniRLTest* adopts a simple reward function, making it challenging to explore apps with complex business logic effectively. Specifically, the reward function overly emphasizes exploring states with more unvisited widgets, which can easily cause infinite scrolling operations. Although *UniRLTest-S* incorporates sensitive APIs into the reward function, it fails to learn the relationship between local widgets and sensitive API calls due to its coarse-grained understanding. The results demonstrate that these methods are not suitable for discovering sensitive APIs.

We can observe that *APIMind* exhibits superior performance compared to *UniRLTest-S* and *UniRLTest*. Specifically, *APIMind* identifies a mean of 2137 sensitive API calls, while *UniRLTest-S* and *UniRLTest* only detect 1495 and 1442, respectively. This suggests that *APIMind* achieves a 43% improvement in efficiency over *UniRLTest*. This is expected because it has the explicit goal of triggering sensitive APIs. With this underlying goal, *APIMind* can automatically learn the relationship between widgets and APIs using neural networks.

Finally, it is worth mentioning the overlap of sensitive APIs identified by the three methods. Indeed, *APIMind* and *UniRLTest-S* find 1222 common sensitive API calls (i.e., the same GUI widgets and contexts). *APIMind* and *UniRLTest* yield 1253 overlaps, while *UniRLTest* and *UniRLTest-S* yield 1081 overlaps. An important finding of this analysis is that *UniRLTest-S* can only identify 104 missed by *APIMind*, whereas *APIMind* detects 924 missed by *UniRLTest-S*. This is mainly due to the consideration of fine-grained features, allowing *APIMind* to identify more sensitive APIs. Besides, *UniRLTest-S* may get stuck in a cycle of constantly clicking a GUI widget that can trigger sensitive APIs, while *APIMind* solves this problem by limiting the visiting frequency of widgets.

C. RQ2: Effectiveness in Assessing Description-to-Permission Fidelity

To evaluate the usefulness of our approach, we perform a consistency analysis between the contexts and collected privacy data. Such analysis helps prevent personal data leakage, such as sending it to third parties without the user's knowledge. To assess the validity of our method, we conduct a comparison with APICOG, which serves as a baseline method. In this study, we execute 121 apps in the testing set and gather accessed activities and sensitive APIs using Trigger. Subsequently, we conduct manual labeling of 2178 $< activity, API >$ pairs for consistency assessment. Lastly, we utilize Fidelity Analyzer and APICOG to analyze each pair and calculate the following metrics, respectively.

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP} \tag{5}$$

$$precision = \frac{TP}{TP + FP} \tag{6}$$

$$recall = \frac{TP}{TP + FN} \tag{7}$$

$$F_1 \ score = \frac{2 \times precision \times recall}{precision + recall} \tag{8}$$

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

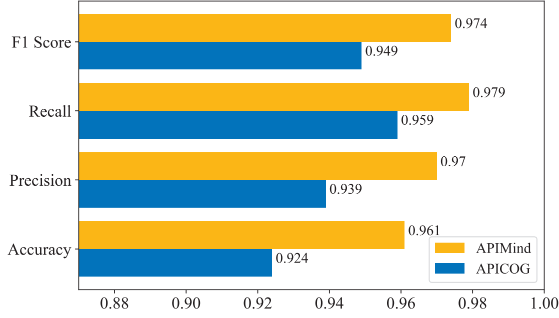

Fig. 6: Performance comparison of *APIMind* and APICOG in assessing description-to-permission fidelity.

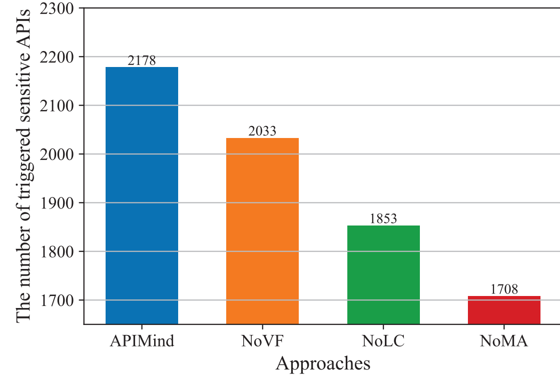

Fig. 7: The ablation analysis of different multimodal features.

where TP, TN, FN, and FP denote true positive, true negative, false negative, and false positive, respectively.

The experimental results indicate that *APIMind* outperforms APICOG in assessing description-to-permission fidelity in Android apps, as shown in Fig. 6. More specifically, the accuracy, precision, recall, and F1 score of *APIMind* are 96.1%, 97.0%, 97.9%, and 97.4%, respectively. APICOG achieves lower results: 92.4%, 93.9%, 95.9%, and 94.9%, respectively. The main reason is that *APIMind* considers the preceding UI context, thus achieving higher detection accuracy. An illustrative example is shown in Fig. 2. The details have been previously introduced in Section II-B. For recall improvement, we also use an example to illustrate this. For instance, an app requests location information in a preceding activity. However, after the request is denied, it still accesses the location information necessary for the current activity. *APIMind* can infer the denial choice from the preceding context and detect the inconsistency, while the baseline cannot. Overall, these results demonstrate the potential usefulness of our tool.

D. RQ3: Effect of Multimodal Features

In this section, we conduct ablation studies of different feature modalities and investigate their effects on model performance. The numerical results are presented in Fig. 7. Here, NoVF, NoLC, and NoMA indicate that we eliminate the visual features, layout context features, and meta-attribute features, respectively. From Fig. 7, we observe that *APIMind* exhibits better performance than NoMA. Specifically, the number of sensitive APIs identified by *APIMind* is much higher than that identified by NoMA, which demonstrates the validity of meta-attributes in sensitive API triggering. The primary reason is that meta-attributes can directly reflect the semantics of GUI widgets, with generalization under different UI contexts.

It can be found that *APIMind* significantly outperforms the variant NoLC. More specifically, the number of sensitive API calls identified by *APIMind* decreases by 325 compared with NoLC. The results imply the effectiveness of layout context features in identifying sensitive API calls. The reason is that the semantics of GUI widgets can be inferred based on the runtime context when meta-attributes, such as texts and content descriptions, are unavailable. For visual features, its impact on model performance is minor relative to the other two. One potential explanation is that considering the GUI state as a whole and understanding it based on vision alone are too coarse-grained to infer the semantics of local widgets.

Overall, the performance of all three variants shows a decline to varying extents. Specifically, *APIMind* discovers 2178 sensitive API calls, while NoVF, NoLC, and NoMA identify 2033, 1853, and 1708, respectively. The results suggest that integrating these feature modalities could enhance the overall performance of *APIMind*. This discrepancy implies that these feature modalities significantly affect model performance, ranked in descending order as meta-attribute features, layout context features, and visual semantic features.

E. RQ4: Prevalence of Typical Inconsistencies

To investigate the prevalence of typical inconsistencies in Android apps, we conduct a large-scale case study using *APIMind*. Our study involves the collection of 1013 realistic Android apps from the Xiaomi App Store from September 26 to October 9, 2022. We choose the Xiaomi App Store because of its popularity and availability of diverse app categories. Similar to the dataset for evaluating *APIMind*, we systematically fetch these apps with IDs ranging from 1000 to 3000. This indicates there is no overlap between the two datasets. To obtain more realistic results, we select 1200 apps using a category-based sampling method and eliminate any that crash during interactions with Frida. Eventually, our dataset includes 1013 apps across 12 categories, such as sports, health, and travel.

Our research, based on an examination of 1013 mobile apps, reveals that 82.8% of them (839 apps) contain at least one inconsistency. Besides, we manually identify several typical inconsistencies. We provide further details below.

Unnecessary Permissions. For 68.7% of inconsistencies, apps collect sensitive data beyond what is necessary for their intended purposes. An illustrative example is provided in Fig. 8(a), which displays a region selection page for a food delivery app. In this instance, the app gathers the device ID, IMSI, phone number, SIM card serial number, and subscriber ID, which are unnecessary for the functionalities they present to users. Note that Android runtime permissions provide

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

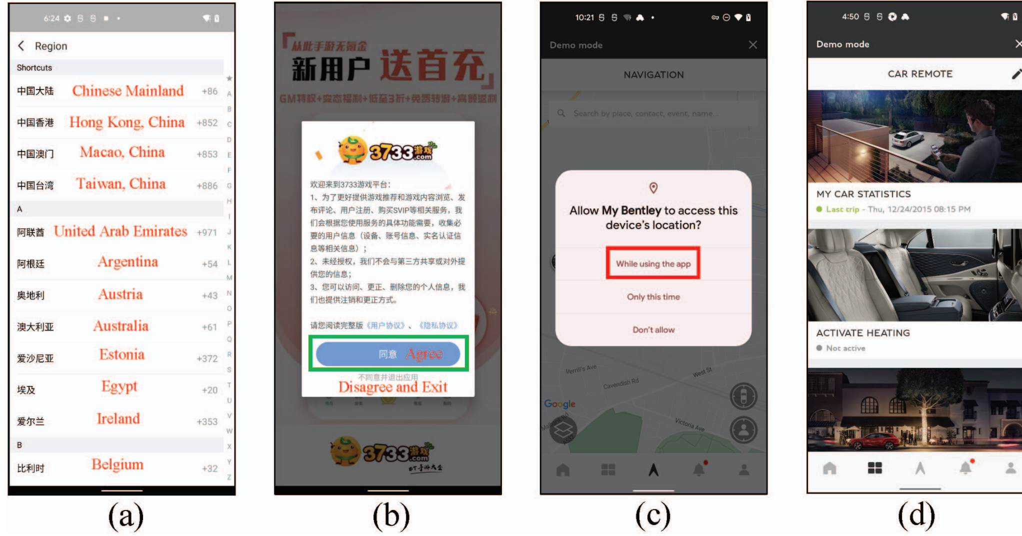

Fig. 8: Some concrete examples of apps with inconsistencies.

insufficient protection for these permissions, exposing users to stealthy data collection practices.

Deceptive Agreements. The second most frequent inconsistency is deceptive agreements, accounting for 27%. In this case, the app may present users with a misleading agreement that appears to offer them the choice of granting requested permissions. However, the app may request additional permissions not disclosed in the user agreement or collect sensitive data it wants without authorization (such as denied or unconsented). For example, as depicted in Fig. 8(b), the *Game Console* app gathered sensitive data before the user agreed to the privacy policy by clicking the green box.

Retaining Permissions. Lastly, for 4.3% of inconsistencies, apps may misuse permissions that have been granted legitimately. For instance, the *My Bentley* app asks for location permission for navigation purposes, which the user approves by selecting the "While using the app" option highlighted by the red box, as shown in Fig. 8(c). However, upon reaching Fig. 8(d), the app goes beyond its scope by abusing the granted permission to gather location data unnecessarily. Such access allows the app to track users' movement continuously, which is an unexpected behavior.

F. RQ5: Responses from the Administrator

To assess the feasibility of our tool, we randomly select 40 apps with inconsistencies and report them to the Personal Information Protection Task Force on Apps (PIPTFoA) for manual assessment. PIPTFoA is a reputable institution created through a joint effort by four authoritative bodies in China: the Ministry of Public Security, the Ministry of Industry and Information Technology, the State Administration for Market Regulation, and the Cyberspace Administration [55]. This institution vigilantly regulates and evaluates mobile app data-gathering practices. Given their limited personnel, we limit our submissions to 40 apps to avoid overwhelming the PIPTFoA team and potential service denial.

As of the paper submission deadline, a total of 29 apps were confirmed for our study. Out of these, eight apps included severe violations, while the remaining 21 had minor infractions. Additionally, 11 apps, apart from those above 29, resolved their issues in the updated version. These findings suggest that our tool exhibits significant potential for assessing description-to-permission fidelity.

TABLE II: The list of apps with severe violations.

Package Name	Version Number
com.jianshu.haruki	6.4.8
com.a3733.gamebox	3.6.1173
com.ddcinemaapp	8.6.8
com.horizon.offer	5.5.18
com.ushaqi.zhuishushenqi	4.85.4
com.baidu.lbs.crowdapp	6.1.6
com.cnki.client	8.5.3
com.daodao.note	1.1.4.1

G. Implications and Suggestions

Our study presents evidence of extensive inconsistencies (82.8% of apps) in Android apps, indicating possible risks of privacy breaches and other security hazards. In addition, our findings identify several common types of inconsistencies, providing meaningful guidance for designing advanced strategies to safeguard privacy.

We provide several suggestions to address these inconsistencies. First, developers should carefully determine the permissions required for each activity during the development process. Second, automated tools like *APIMind* should be designed and developed to detect such inconsistent behaviors. Third, access control should be more granular, especially at the activity level. This indicates that each activity cannot directly access permissions granted in other activities and must request them again if necessary. Finally, permission dialogs

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

should include detailed explanations. This enables users to comprehend better what permissions the app requests.

V. Discussions

A. Threats to Validity

Internal Threats. The hyperparameter setting poses the primary threat to internal validity. Due to limitations of time and resources, the exhaustive fine-tuning of hyperparameters is infeasible, implying that the current configuration might not be optimal. To tackle this issue, we strive to follow the precedent of past research. Additionally, for cases where there is no known reference, small-scale experiments are designed to ensure their best practices.

External Threats. The main threat to external validity is the limited sample size of apps tested in the experiments. To address this threat, we implement a systematic method to build the dataset, which involves retrieving apps from the Xiaomi app market using an auto-incremental ID allocation. While evaluating a larger pool of apps would be ideal, our sample includes a large and diverse set of apps from various categories, suggesting the general applicability of our methodology. Another potential threat is the bias in manually annotating $< activity, API >$ pairs. However, prior research [56] indicates that senior students can act as reliable proxies in a controlled setting.

B. Limitations

Despite the promising performance of our tool, it still suffers from some limitations. First, *APIMind* instruments apps with Frida to monitor sensitive APIs, but it only supports native code and Java bytecode, not non-Java code. However, Frida is replaceable, and *APIMind* can integrate with future tools for non-Java code. Second, *APIMind* uses dynamic analysis, which may fail to access or test login functionalities. This may limit the applicability of our approach. Third, *APIMind* only employs DQN-based techniques. We plan to explore other RL architectures in future work.

VI. Related Work

GUI Testing. Random methods, such as Monkey [22] and Dynodroid [57], were used in the earliest stages to automate mobile app testing. To improve test comprehensiveness, some scholars have investigated utilizing state machines [54], [58], [59] or systematic strategies [60], [61] to produce superior test cases. There have been studies that deploy machine learning in GUI testing [25], [27], [31], [62]–[65]. Currently, reinforcement learning techniques are prevalent in testing tasks [25], [27], [62], [66]. Zhang *et al.* [53] proposed *UniRLTest*, a curiosity-driven reinforcement learning framework to guide the exploration of unfamiliar functions. Nonetheless, these testing frameworks are designed to achieve maximum code or GUI coverage in general but are ineffective in discovering specific targets like sensitive API calls. Unlike these general techniques, *APIMind* is a dynamic approach that understands GUI widgets at a fine-grained level and correlates API calls with dual UI contexts (i.e., preceding and current contexts).

NLP Techniques for Android. NLP techniques have been employed to extract semantics from textual descriptions to perform the security analysis of mobile apps [2]–[4], [15], [16], [67]–[74]. For instance, Qu *et al.* [3] proposed an end-to-end tool AutoCog to detect whether the requested permissions are consistent with textual descriptions of apps. Yu *et al.* [4] proposed to incorporate privacy policies to enhance detection performance. However, these methods are too coarse-grained to determine the consistency between the API-usage purposes and the app's actual behavior.

Moreover, Pan *et al.* proposed FlowCog [16], a system that can analyze the semantics of information flows and detect whether they are justified by app behaviors. However, static analysis methods tend to be practically infeasible and undesirable due to the intense use of dynamic features such as code obfuscation, code encryption, dynamic class loading, and reflection (especially in malware apps). APICOG [2] is a dynamic analysis framework that can check the legitimacy of sensitive API calls by correlating them with their runtime UI contexts and extracting natural language semantics from both the app and API documentation.

VII. Concluding Remarks

In this paper, we present *APIMind*, a novel automated tool for assessing runtime description-to-permission fidelity in Android apps. The key idea behind it is to efficiently trigger sensitive APIs by utilizing multimodal features to jointly infer the semantics of GUI widgets and leveraging deep networks to automatically learn their relationship. Furthermore, it can accurately examine the legitimacy of sensitive API calls by developing an extended version of an existing tool that considers dual UI contexts (i.e., preceding and current contexts). Based on our newly proposed automated testing tool, we conduct a large-scale case study of 1013 real Android apps. Our finding reveals the prevalence of several typical inconsistencies: unnecessary permissions, deceptive agreements, and retaining permissions. Our work indicates that API-driven methods can help detect more permission-related issues, which shows a promising direction for privacy-related research for mobile apps.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the National Natural Science Foundation of China (62141208, 62172009).

References

[1] COUNTERPOINT, "Market share for smartphone oses." 2022. [Online]. Available: https://www.counterpointresearch.com/global-smartphone-share/

[2] J. Liu, D. He, D. Wu, and J. Xue, "Correlating ui contexts with sensitive api calls: Dynamic semantic extraction and analysis," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 241–252.

[3] 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*, 2014, pp. 1354–1365.

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

[4] L. Yu, X. Luo, C. Qian, S. Wang, and H. K. Leung, "Enhancing the description-to-behavior fidelity in android apps with privacy policy," *IEEE Transactions on Software Engineering*, vol. 44, no. 9, pp. 834–854, 2017.

[5] O. Olukoya, L. Mackenzie, and I. Omoronyia, "Security-oriented view of app behaviour using textual descriptions and user-granted permission requests," *Computers & Security*, vol. 89, p. 101685, 2020.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[7] P. A. M. Al-Bakri and H. L. Hussein, "Static analysis based behavioral api for malware detection using markov chain," *The International Institute for Science, Technology and Education (IISTE)*, vol. 5, no. 2014, 2014.

[8] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware detection based on mining api calls," in *Proceedings of the 2010 ACM symposium on applied computing*, 2010, pp. 1020–1025.

[9] Y. Liu, N. Xi, and Y. Zhi, "Nleu: A semantic-based taint analysis for vetting apps in android," in *2021 International Conference on Networking and Network Applications (NaNA)*. IEEE, 2021, pp. 327–333.

[10] Y. Li, Y. Guo, and X. Chen, "Peruim: Understanding mobile application privacy with permission-ui mapping," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2016, pp. 682–693.

[11] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 209–220.

[12] H. Wang, J. Hong, and Y. Guo, "Using text mining to infer the purpose of permission use in mobile apps," in *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing*, 2015, pp. 1107–1118.

[13] D. Liang, R. Chen, and H. Sun, "Droidmonitor: a high-level programming model for dynamic api monitoring on android," *Proc. NSCE. CRC Press*, pp. 93–96, 2014.

[14] W. Fan, Y. Sang, D. Zhang, R. Sun, and Y. Liu, "Droidinjector: A process injection-based dynamic tracking system for runtime behaviors of android applications," *Computers & Security*, vol. 70, pp. 224–237, 2017.

[15] H. Wang and Y. Guo, "Understanding third-party libraries in mobile app analysis," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 515–516.

[16] 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 18)*, 2018, pp. 1669–1685.

[17] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 331–341.

[18] H. Wang, Y. Guo, Z. Tang, G. Bai, and X. Chen, "Reevaluating android permission gaps with static and dynamic analysis," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

[19] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhut, G. Li, and J. Wang, "Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 322–334.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[21] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, 2011, pp. 77–83.

[22] Google, "Ui/application exerciser," 1983. [Online]. Available: http://developer.android.com/tools/help/monkey.html

[23] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 599–609.

[24] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana, "Agrippin: a novel search based testing technique for android applications," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, 2015, pp. 5–12.

[25] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Transactions on Software Engineering and Methodology*, 2022.

[26] R. Tufano, S. Scalabrino, L. Pascarella, E. Aghajani, R. Oliveto, and G. Bavota, "Using reinforcement learning for load testing of video games," *arXiv preprint arXiv:2201.06865*, 2022.

[27] E. Collins, A. Neto, A. Vincenzi, and J. Maldonado, "Deep reinforcement learning based android application gui testing," in *Brazilian Symposium on Software Engineering*, 2021, pp. 186–194.

[28] F. YazdaniBanafsheDaragh and S. Malek, "Deep gui: black-box gui input generation with deep learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 905–916.

[29] B. Jiang, W. Wei, L. Yi, and W. Chan, "Droidgamer: Android game testing with operable widget recognition by deep learning," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 197–206.

[30] J. Eskonen, J. Kahles, and J. Reijonen, "Automating gui testing with image-based deep reinforcement learning," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 160–167.

[31] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 105–115.

[32] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 2–8.

[33] F. Y. B. Daragh and S. Malek, "Deep gui: Black-box gui input generation with deep learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 905–916.

[34] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[35] S. Zhang, Y. Li, W. Yan, Y. Guo, and X. Chen, "Dependency-aware form understanding," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 139–149.

[36] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[37] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[38] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[39] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[40] M. Chen, Z. Chang, H. Lu, B. Yang, Z. Li, L. Guo, and Z. Wang, "Augnet: End-to-end unsupervised visual representation learning with image augmentation," 2021.

[41] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017, pp. 845–854.

[42] B. Wang, G. Li, X. Zhou, Z. Chen, T. Grossman, and Y. Li, "Screen2words: Automatic mobile ui summarization with multimodal learning," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021, pp. 498–510.

[43] S. Feiz, J. Wu, X. Zhang, A. Swearngin, T. Barik, and J. Nichols, "Understanding screen relationships from screenshots of smartphone applications," in *27th International Conference on Intelligent User Interfaces*, 2022, pp. 447–458.

[44] T. J.-J. Li, L. Popowski, T. Mitchell, and B. A. Myers, "Screen2vec: Semantic embedding of gui screens and gui components," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–15.

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.

[45] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, pp. 9–44, 1988.
[46] Stanford, "[online]." 2010. [Online]. Available: https://github.com/dasmith/stanford-corenlp-python
[47] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques." in *NDSS*, 2016.
[48] OpenATX, "Uiautomator2," 2017. [Online]. Available: https://github.com/openatx/uiautomator2/releases
[49] Facebook, "Pytorch," 2016. [Online]. Available: https://github.com/pytorch/pytorch
[50] Oleavr, "Frida," 2016. [Online]. Available: https://github.com/frida/frida/releases
[51] L. Gong, Z. Li, F. Qian, Z. Zhang, Q. A. Chen, Z. Qian, H. Lin, and Y. Liu, "Experiences of landing machine learning onto market-scale mobile malware detection," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–14.
[52] Y. Wu, D. Zou, W. Yang, X. Li, and H. Jin, "Homdroid: detecting android covert malware by social-network homophily analysis," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 216–229.
[53] Z. Zhang, Y. Liu, S. Yu, X. Li, Y. Yun, C. Fang, and Z. Chen, "Unirltest: universal platform-independent testing with reinforcement learning via image understanding," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 805–808.
[54] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.
[55] C. government, "Announcement on carrying out special governance on illegal and irregular collection and use of personal information by app," 2019. [Online]. Available: http://www.gov.cn/zhengce/zhengceku/2019-11/11/content_5450754.htm
[56] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *2015 IEEE/ACM 37th IEEE international conference on software engineering*, vol. 1. IEEE, 2015, pp. 666–676.
[57] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.
[58] 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*, 2017, pp. 245–256.
[59] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.
[60] 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*, 2016, pp. 94–105.
[61] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660.
[62] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.
[63] N. P. Borges Jr, M. Gómez, and A. Zeller, "Guiding app testing with mined interaction models," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 2018, pp. 133–143.
[64] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 42–53.
[65] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1070–1073.
[66] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 12–22.
[67] C. Zhang, H. Wang, R. Wang, Y. Guo, and G. Xu, "Re-checking app behavior against app description in the context of third-party libraries." in *SEKE*, 2018, pp. 665–664.
[68] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "{WHYPER}: Towards automating risk assessment of mobile applications," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 527–542.
[69] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1025–1035.
[70] 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 *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1036–1046.
[71] F. Dong, H. Wang, L. Li, Y. Guo, G. Xu, and S. Zhang, "How do mobile apps violate the behavioral policy of advertisement libraries?" in *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, 2018, pp. 75–80.
[72] Y. Li, F. Chen, T. J.-J. Li, Y. Guo, G. Huang, M. Fredrikson, Y. Agarwal, and J. I. Hong, "Privacystreams: Enabling transparency in personal data processing for mobile apps," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 3, pp. 1–26, 2017.
[73] Y. Hu, H. Wang, T. Ji, X. Xiao, X. Luo, P. Gao, and Y. Guo, "Champ: Characterizing undesired app behaviors from user comments based on market policies," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 933–945.
[74] M. Liu, H. Wang, Y. Guo, and J. Hong, "Identifying and analyzing the privacy of apps for kids," in *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*, 2016, pp. 105–110.

Authorized licensed use limited to: Peking University. Downloaded on December 12,2023 at 03:00:46 UTC from IEEE Xplore. Restrictions apply.