2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00141

How Android Apps Break the Data Minimization Principle: An Empirical Study

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

***Abstract*—The Data Minimization Principle is crucial for protecting individual privacy. However, existing Android runtime permissions do not guarantee this principle. Moreover, the lack of an automatic enforcement mechanism leads to uncertainty as to whether apps strictly comply with this principle. To bridge this gap, we conduct the first systematic empirical study on violations of the Data Minimization Principle and design a new enforcement tool called *GUIMind* to detect them. *GUIMind* first utilizes a reinforcement learning model to explore app activities and monitor access to sensitive APIs that require sensitive permissions, and then it leverages an existing tool to detect such violations. We evaluate the performance of *GUIMind* using 120 real-world Android apps. The results indicate that *GUIMind* can achieve a detection accuracy of 96.1%, effectively accelerating the empirical study. Our empirical research is mainly focused on the prevalence of violations, the responses of administrators to violations, and the potential factors and characteristics that lead to violations, such as typical violations, app categories, and personal data types. Our study reveals that 83.5% of apps contain at least one privacy violation, with health apps being the most severe. In addition, telephony information is the most commonly leaked personal data type, accounting for 71.1%. Finally, we randomly selected 60 non-compliant apps for reporting to the administrator, whose responses confirm the effectiveness of our approach.**

I. Introduction

The Data Minimization Principle is a key privacy principle that aims to respect individual privacy and reduce the risks of data breaches. It means that personal information should only be collected, stored, and used if it is directly relevant and necessary for a specific purpose. It also means that personal information should not be kept longer than needed for that purpose unless it is for statistical purposes. This principle is embedded in various legal frameworks, such as the EU's General Data Protection Regulation (GDPR) [1], Brazil's Lei Geral de Proteção de Dados Pessoais (LGPD) [2], the California Privacy Rights Act (CPRA) in the United States [3], the Data Protection Act in the United Kingdom [4], and the Personal Information Protection Law (PIPL) in China [5]. However, despite its importance, this principle only exists in legal provisions and is not guaranteed by technical means. Although the Android runtime permission system aims to uphold the Data Minimization Principle by enforcing a popup window before accessing sensitive permissions, recent studies revealed that this system is ineffective in practice [6]–[9]. Therefore, we anticipate that many apps will violate the Data Minimization Principle due to the lack of effective enforcement techniques.

Unfortunately, although we suspect there are a lot of risks in violating the Data Minimization Principle, currently, there is no empirical evidence to support this suspicion. To this end, we conduct the first systematic measurement study on violations of the Data Minimization Principle. In our study, we answer five research questions: *what are the types of typical violations of the Data Minimization Principle, how prevalent are privacy non-compliance issues among apps, how do privacy non-compliance issues occur in different categories of apps, which types of personal data are most frequently leaked by apps, and what is the response from the administrators to the violations?* These research questions help us understand violations of the Data Minimization Principle in practice. In addition, our research findings provide valuable insights into developing effective strategies to ensure privacy compliance in apps and increasing developer and user awareness of the importance of the Data Minimization Principle.

Although it is valuable to empirically measure the violations of the Data Minimization Principle, it is particularly challenging to build an automated testing tool that can detect such violations in a large number of apps due to (1) the lack of a formal definition; and (2) the difficulty of discovering sensitive data across complex and dynamic app environments. To address these challenges, in this paper, we first formally define the violations of the Data Minimization Principle. Then, we propose a novel tool called *GUIMind* to automatically discover accesses to sensitive permissions in market apps. *GUIMind* consists of two major components. The first component is Explorer, which uses a deep reinforcement learning-based model to explore app activities and monitor the accesses to sensitive APIs that require sensitive permissions. The second component is Fidelity Checker, which uses the existing tool APICOG [10] to check whether an activity collects more sensitive permissions than the users expect. We evaluate the performance and accuracy of *GUIMind* with 120 real-world apps from the Xiaomi market. The results show that *GUIMind* can accelerate the empirical study while achieving a detection accuracy of 96.1%.

Based on our newly proposed automated testing tool, we conduct a large-scale empirical study of 1,876 real Android

†Corresponding authors.

apps. Our findings reveal four typical violations of the Data Minimization Principle: irrelevant permissions, holding permissions, bogus agreements, and mismatched permissions with the agreement. Shockingly, our study shows that 83.5% of apps have at least one privacy violation, with health apps performing the worst. Furthermore, the most frequently leaked personal data type is telephony information, accounting for 71.1%. To further validate the effectiveness of our tool, we randomly select 60 apps that violate the Data Minimization Principle and report them to the administrator. The response received from the administrator confirms the effectiveness of our approach.

The result of our measurement study indicates the significant prevalence of violations of the Data Minimization Principle for Android apps in various domains and scenarios. Our study reveals the diversity and specificity of privacy violations in the Android app ecosystem and calls for more attention and efforts from developers, users, and regulators to protect personal data from being unnecessarily collected, stored, or transmitted. We also demonstrate the effectiveness and usefulness of our proposed automated testing tool, which can help app developers identify and fix privacy violations in their apps and help users and regulators audit and monitor the privacy practices of apps. Our tool can also facilitate further research on app privacy analysis and improvement.

We summarize our contribution as follows.

- We formally define the violations of the Data Minimization Principle.
- To the best of our knowledge, we conduct the first systematic empirical study on violations of the Data Minimization Principle in mobile apps. This work benefits developers/regulators in detecting privacy violations and market-level reviews of app stores.
- We propose a novel automated tool called *GUIMind* to detect violations of the Data Minimization Principle, which leverages a deep reinforcement learning-based approach to prioritize and explore activities with the explicit goal of triggering sensitive API calls that require sensitive permissions.
- We thoroughly evaluate our approach with a variety of Android apps collected from the Xiaomi Store. The experimental results demonstrate the effectiveness of *GUIMind* in detecting violations of the Data Minimization Principle.

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

II. Background And Problem Statement

A. Data Minimization Principle

The Data Minimization Principle is a fundamental principle of data protection. It requires that organizations collect, process, and store only the minimum amount of personal data necessary to achieve specific and legitimate purposes. This principle is enshrined in various data protection frameworks. In the European Union, the GDPR [1] enshrines the Data Minimization Principle as one of its core principles, stating that personal data shall be "adequate, relevant, and limited to what is necessary in relation to the purposes for which they are processed." Similarly, Brazil's LGPD [2] also requires data controllers to limit the collection and processing of personal data to what is strictly necessary for a specific purpose. In the United States, the recently enacted CPRA [3] includes provisions on data minimization, requiring businesses to limit their collection, use, and disclosure of personal information to what is reasonably necessary to achieve their stated purpose. The UK's Data Protection Act [4] also requires that data controllers observe the Data Minimization Principle while processing personal data to ensure the gathering of only necessary and relevant data. In China, the PIPL [5] emphasizes the Data Minimization Principle by requiring organizations to minimize the collection of personal information and to anonymize or pseudonymize personal information wherever possible. Overall, the Data Minimization Principle is a critical component of data protection frameworks worldwide, ensuring that personal data is collected, processed, and stored only to the extent necessary for legitimate purposes.

B. Limitations of the Android Permission System

The Android permission system is designed to control app access to device resources and data. While it is intended to protect user privacy and ensure data security, it cannot fully guarantee the Data Minimization Principle. There are two types of Android permissions: static permissions and runtime permissions. For static permissions, users are required to grant permissions during installation. This indicates that the app can access specified resources without further user actions. While this level of access may be necessary to function properly for apps, it also poses a risk to user privacy. For example, some apps may request more permissions than necessary, leading to unnecessary data collection and potentially violating the Data Minimization Principle.

Runtime permissions are introduced in Android 6.0 to address some of these privacy concerns [11]. With runtime permissions, the user is prompted to grant or deny access to specific resources when the app requests it rather than during installation. Ideally, if users can deny unnecessary permission requests, it ensures only the minimum amount of data necessary is collected. However, recent studies show [6]–[9] that Android runtime permissions are not sufficient to fully prevent apps from abusing users' sensitive data due to three reasons: (1) apps may group different permissions and mislead users into granting unnecessary permissions; (2) the majority of users (54.7%) *cannot* understand the descriptions in the message box for permissions, allowing apps to access sensitive data without true consent from users; (3) many sensitive permissions, such as device location or unique phone IDs (e.g., IMEI number), are not protected.

In summary, while the Android Permission System provides some control over app access to user data and resources, it cannot guarantee adherence to the Data Minimization Principle. Further efforts are needed to ensure that apps only collect and retain the minimum user data necessary for legitimate purposes.

C. Android GUI Programming

In Android development, activities are the building blocks of apps. An activity represents a single screen with a user interface. It typically fills the entire screen, although smaller activities can be used in multi-window mode or as floating windows. Widgets are small GUI components that allow users to interact with an app, such as buttons, text fields, checkboxes, and radio buttons. Activities are responsible for managing the presentation and behavior of their widgets.

A frame GUI window is a container that holds one or more activities. It provides a framework for the activities to be displayed and allows them to communicate with each other. An activity can be launched from another activity using an Intent object, and the system handles the transition between the two activities. A GUI transition graph is a diagram that represents the behavior and functionality of an app. Formally, we define a GUI transition graph as a directed graph $G = (V, E)$ where V is the set of nodes in the graph, representing individual activities or screens within the app, and E is the set of edges in the graph, representing the possible transitions or actions between those activities. It provides a high-level overview of the flow and structure of the app, showing how users can navigate between different screens and what actions or events trigger these transitions. This graph is useful when designing and testing the flow of an app's user interface.

D. Assumptions

In this paper, we hold a "sloppy user" assumption, which assumes that users do not have sufficient knowledge or patience to carefully examine the permissions required by apps. Thus, there is a high probability for users to mistakenly grant sensitive permissions to apps, sometimes even unconsciously.

This assumption is supported by recent studies [6]–[8]. Furthermore, there are also many reports and blogs on the Internet that supports the "sloppy user" assumption. For instance, a study by Carnegie Mellon University [12] reveals that many users grant app permissions without reading the prompts carefully. A survey of users with an average of 5 years of Android experience [13] indicates that nearly 40% of the participants fail to make informed decisions to grant permissions due to an inadequate understanding of permission prompts. Additionally, both the Internet survey and laboratory study [14] show that users are less concerned about Android permissions.

E. Problem Statement

On a high level, we say an app violates the Data Minimization Principle if it *attempts* to collect, use, or hold permissions that are not necessary to the user-expected functionalities of the current activity. Note that, in this paper, the violations have two properties:

- Intention Based: we *do not* require the app to successfully get the permission(e.g., bypassing the runtime permission systems of Android). As long as the app tries to obtain unnecessary permissions, we consider it a violation. This property is valid since the users often mistakenly grant unnecessary runtime permissions to apps, according to the "sloppy user" assumption.
- Fine-Grained: we detect violations on the granularity of the activity[1] level, instead of on the app level. Traditional app-level detection of privacy violations may overlook specific instances of permission abuse or unauthorized access within the app. By detecting activity-level violations, users gain a more granular understanding of how their data is used. For example, suppose a news app requests location information to recommend top local news. While this may seem harmless, using this access to collect and store location data for other unnecessary purposes would violate the Data Minimization Principle.

The formal definition of a violation of the Data Minimization Principle is given as follows. We define an Android app as a set that has n activities: $A = \{a_1, a_2, a_3, ...a_n\}$. For each activity a_i, we define it as a tuple of $a_i =< B_i, P_i^c >$, where $B_i = \{b_{i1}, b_{i2}, ..., b_{in}\}$ is the set of user-expected behaviors of the activity and $P_i^c = \{p_{i1}^c, p_{i2}^c, ..., p_{in}^c\}$ is a set of permissions that the activity attempts to get. Here we say a behavior b_{ij} is user-expected if it has been clearly described by the GUI context of the corresponding activity a_i. We further define a map $R := B \longrightarrow P$ that maps a behavior b_i to the required permissions $p \in P$ to satisfy the user expectation. For simplicity, we define $R(B)$ as

$$\bigcup_{b_i \in B} R(b_i)$$

, then we say there is a violation of the Data Minimization Principle in activity a_i if $P_i \nsubseteq R(B_i)$.

III. Design of *GUIMind*

To automatically discover violations of the Data Minimization Principle, we develop *GUIMind*, a novel automated tool that detects inconsistencies between GUI contexts and permissions for activities in Android apps. The workflow of *GUIMind* is shown in Figure 1. *GUIMind* takes an APK file as input and generates a report containing any violations of the Data Minimization Principle during the execution. It has two components. The first is Explorer, which thoroughly explores activities in the target app and monitors the accesses to API calls that attempt to get sensitive permissions (sensitive APIs). The second is Fidelity Checker, which is used to detect non-compliance between user-expected behaviors and access to sensitive APIs.

1) Explorer: Explorer accepts an Android app and thoroughly explores the activities in the app. During the exploration, Explorer also outputs the screenshots, layout files, and permissions of the activities to the Fidelity Checker to detect violations of the Data Minimization Principle.

[1]An activity represents a single screen with a user interface. It is the most basic functional unit of the Android system [15]. An app may contain multiple activities.

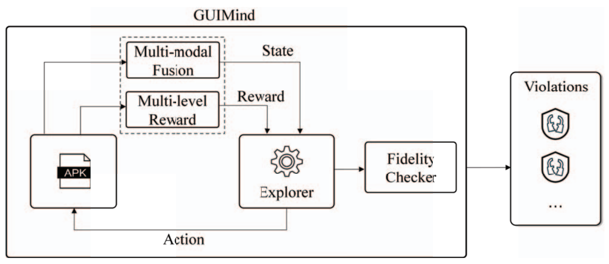


Fig. 1: Overview of *GUIMind*.

The key challenge of Explorer is how to efficiently trigger sensitive API calls that require sensitive permissions. The straightforward approach is to utilize a GUI tester [16] that generates GUI operations randomly [17] or based on heuristics [18], [19]. However, existing GUI testers are not optimized for exploring sensitive API calls. Thus, they may waste time exploring activities that do not require sensitive permissions and slow down our empirical study. To this end, we model Explorer as a Reinforcement Learning(RL) agent with a deep learning model that automatically prioritizes activities based on their probability of invoking sensitive APIs. Thus, Explorer can focus on activities related to sensitive APIs and reduce the time overhead of our empirical study.

Specifically, Explorer is an RL agent that uses deep Q-Network (DQN) [20] to navigate through the transition graph of all the activities in an Android app. The core part of Explorer is the Q-Network, which takes as input the current state and outputs the expected reward for each possible action. The structure of the Q-Network consists of a feedforward neural network. This is reasonable since it can approximate any complex functions [21], which is useful for modeling the relationships between activities and sensitive APIs.

The architecture of the Q-Network consists of three fully connected layers, as shown in Equation 1.

$$f(x) = W_3\sigma(W_2\sigma(W_1x + b_1) + b_2) + b_3 \tag{1}$$

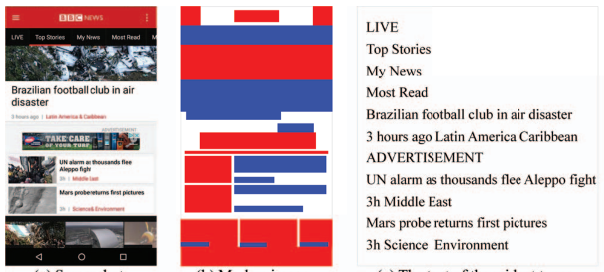
where W_* are weight matrices, b_* are bias vectors, σ is the activation function, and x is a state representation described as a

Algorithm 1: The Explorer Learning Process.

```
1  Initialize DQN model $M$;
2  Initialize replay memory $\mathcal{D}$;   // Store past samples
3  $p_0 \leftarrow$ launchApp();   // Get the first UI page
4  $s_0 \leftarrow$ fuseMultiModal($u_0$);   // Multi-modal fusion
5  for $t$ = $0,T$ do
6      $a_t \leftarrow$ selectAction($s_t$,$M$);
7      $p_{t+1} \leftarrow$ execute($a_t$);
8      $r_t \leftarrow$ calcMultiLevelReward();   // Eq. 2
9      $s_{t+1} \leftarrow$ fuseMultiModal($p_{t+1}$);
10     $\mathcal{D} \leftarrow \mathcal{D} \cup$ ($s_t$, $a_t$, $r_t$, $s_{t+1}$);   // Store transition
11     $B \leftarrow$ sampleMiniBatch($\mathcal{D}$);
12     $M \leftarrow$ updateModel($B$,$M$);
13     $s_{t+1} \leftarrow s_t$;
```

Multi-modal Fusion of different types of inputs. Specifically, for an activity, the Q-network accepts three types of inputs: the screenshot of the activity, the markup image, and the attributes of each widget. We send the screenshot to an unsupervised visual representation network AugNet [22] that generates a feature vector with 768 dimensions. We send the markup image to a self-supervised embedding model LayoutAutoEncoder [23], generating a feature vector with 64 dimensions. We send the attributes to a pre-trained multilingual sentence embedding model SentenceTransformer [24] that generates a feature vector with 768 dimensions. Then we concatenate the three feature vectors with a concat-attention layer [25]. Finally, the output layer is a dense layer with 768 dimensions. Note that we use SentenceTransformer to extract widget attribute features because it can encode text from different languages into a shared semantic space, thus minimizing the impact of language differences on model performance.

We have three distinct types of input to accurately pinpoint widgets that may lead to other activities. Intuitively, we only need to feed the screenshot of an activity to a deep learning model, which will automatically pinpoint the clickable widgets. However, such an end-to-end solution requires a large model and a huge amount of data, which is not applicable to our study. Therefore, we add a markup image of the screenshot, which explicitly pinpoints the positions of clickable widgets, along with the screenshot and the text of the widget types. An example of a screenshot, its corresponding markup image, and the text of the widget types is shown in Figure 2. Note that we can automatically get the markup image and the text of the widget types by parsing the layout XML file of an Android app with existing tools [23], [26].


LIVE
Top Stories
My News
Most Read
Brazilian football club in air disaster
3 hours ago Latin America Caribbean
ADVERTISEMENT
UN alarm as thousands flee Aleppo fight
3h Middle East
Mars probe returns first pictures
3h Science Environment

(a) Screenshot (b) Markup image (c) The text of the widget types

Fig. 2: An example of different types of input. The "Makeup image" refers to assigning different colors to individual widgets within a GUI screenshot according to their categories (e.g., text, image, and background). "The text of the widget types" refers to the "text" attribute of the widget.

The goal of the Q-Network is to predict the probability of future activities accessing sensitive APIs. To achieve this goal, we need to achieve two sub-goals: (1) exploring more activities containing sensitive APIs; (2) exploring more new activities. Therefore, we design a **Multi-level Reward** simultaneously considering the two sub-goals. The Multi-level Reward consists of three parts: the API-level reward, the activity-level reward, and the widget-level reward. Particularly, the API-level reward is to achieve the first sub-goal of prioritizing activities with more sensitive API calls. In contrast, the other two rewards are used to achieve the second sub-goal of exploring more new activities. Formally, we define the Multi-level Reward as:

$$R_i = \sum_{k \in \{a,s,w\}} \gamma^k R_i^k \tag{2}$$

where R_i^a, R_i^s, and R_i^w represent the API-level reward, activity-level reward, and widget-level reward, respectively, and γ^* is a weighting factor that provides the relative importance of each item.

For the API-level reward R_a, we define it as the number of distinct sensitive APIs called by an activity. We define the activity-level reward as Equation 3, which helps the Explorer deprioritize similar activities. Specifically, in Equation 3, X is the screenshot of the current activity, Y is the screenshot of the successor of X in the GUI transition graph, δ denotes the similarity threshold, AUG is the embedded model AugNet [22] that converts the screenshots to numerical vectors, $sim(\cdot,\cdot)$ represents the Euclidean metric, and $\mathcal{I}$ stands for the indicator function.

$$R_i^s = \mathcal{I}(sim(AUG(X), AUG(Y)) < \delta) \tag{3}$$

For the widget-level reward, we define it as Equation 4, where VF_{cw} denotes the visiting frequency of current widgets, UW_{ns} stands for the number of unvisited widgets in the next GUI state, and M and N represent the scaling factors that ensure comparability between the two.

The rationale behind the widget-level reward is that the transitions between activities are often caused by operating widgets (e.g., clicking a button). Therefore, an activity with more unvisited widgets is more likely to transit to new activities. Thus, we combine the widget-level reward with the activity-level reward to better lead the Explorer to new activities.

$$R_i^w = \frac{M}{VF_{cw}} + \frac{UW_{ns}}{N} \tag{4}$$

2) Fidelity Checker: Fidelity Checker evaluates whether an activity collects more sensitive permissions than the users' expectations. The input is a pair $< activity, API >$, and the output is a binary label that indicates whether the semantics of the API match the GUI context of the activity.

We directly use APICOG [10] to determine whether the GUI context of an activity is consistent with its collected permissions. The high-level idea of APICOG [10] is as follows. First, it uses the Tesseract OCR engine [27] to extract embedded texts in GUI screenshots. Then, it inputs text extracted from screenshots, text-typed attributes of GUI widgets, app descriptions, and descriptions of the called sensitive APIs to a machine learning model to determine whether they are consistent. Due to the page limit, we omit the details about APICOG in this paper.

IV. Evaluation of *GUIMind*

To assess the effectiveness of *GUIMind*, we evaluate its accuracy in detecting violations of the Data Minimization Principle and the speed to trigger the usage of sensitive APIs.

1) Experiment Setup: We conduct experiments on an Ubuntu server 18.04 LTS with a CUDA-enabled Nvidia GTX 1080 Ti GPU of 11GB memory. All apps run on a real rooted mobile device with Android, namely Google Pixel 5. We do not choose an emulator because it prevents sensitive data leakage if malware finds itself in an emulator environment [28].

GUIMind dynamically explores the instrumented apps via Uiautomator2 [26]. It is built and implemented using the Pytorch framework [29]. *GUIMind* uses Frida [30] to automatically instrument apps to monitor sensitive API calls. For dynamic API monitoring, we write snippets of JavaScript, inject them into apps, and interact with the Frida server.

According to previous research [31], we set a maximum of 20 minutes to explore each app. Moreover, in our implementation, we set the weighting factors of the API-level reward, activity-level reward, and widget-level reward to 50, 2, and 1, respectively. For the widget-level reward, the scaling factors M and N are set to 2 and 5, respectively. The similarity threshold δ is set to 2. All hyperparameters are determined based on best practices from a small-scale experiment.

We adopt a systematic approach to build the dataset used in the experiments. Specifically, we implement a crawler to fetch apps from the Xiaomi app store (https://app.mi.com/download/{id}). The placeholder "id" ranges from 1 to 1000. After removing invalid links, the dataset contains a total of 520 apps. We use 400 apps for training and the left 120 apps for testing. There is a broader range of app sizes from 128KB to 424 MB.

2) Accuracy: Although the detection accuracy of our Fidelity Checker is the same as the existing tool [10], we still empirically evaluate its accuracy to ensure the internal validity of our study. To this end, we run the 120 apps in the testing set and collect the visited activities and sensitive APIs through Explorer. Then, we recruit three senior students to manually label 2178 $< activity, API >$ pairs for whether they are consistent. A pair is deemed inconsistent only if at least half of the participants support this claim. Finally, for each pair, we input it to the Fidelity Checker and calculate the accuracy, precision, recall, and F1 score. We show the data in Table I, in which we also include the accuracy numbers from the APICOG paper [10].

The experiment shows that *GUIMind* can achieve an accuracy of 96.1%, a precision of 97.0%, a recall of 97.9%,

TABLE I: Overall performance of *GUIMind* and APICOG in detecting violations of the Data Minimization Principle.

Model	Accuracy	Precision	Recall	F1 Score
APICOG	97.7%	94.1%	92.8%	93.4%
GUIMind	96.1%	97.0%	97.9%	97.4%

and an F1 score of 97.4%, while APICOG yields an accuracy of 97.7%, a precision of 94.1%, a recall of 92.8%, and an F1 score of 93.4%. These numbers are consistent with the numbers reported in the APICOG paper [10]. Overall, these results demonstrate that *GUIMind* has sufficient accuracy to ensure the soundness of statistical results in our empirical study.

In addition, we also observe some discrepancies between the performance of *GUIMind* and APICOG. Specifically, compared to APICOG, *GUIMind* exhibits lower accuracy but higher precision, recall, and F1 scores. A potential reason is that our dataset contains a more balanced sample distribution than the one used in the APICOG paper.

3) Efficiency: In this section, we evaluate whether Explorer can accelerate the empirical study. To this end, we compare its speed of invoking sensitive APIs to two baseline methods. One is a recently proposed RL GUI testing framework (*UniRLTest*) [32] that aims solely to maximize coverage. Note that UniRLTest is a more recent and efficient GUI tester than Droitbot [33], the GUI tester of APICOG. The second baseline(*UniRLTest-S*) is an enhanced *UniRLTest* that considers the number of sensitive APIs in each activity. We implement *UniRLTest-S* by adding the number of sensitive APIs to the target function of *UniRLTest*. The second baseline evaluates the effectiveness of our design of the Q-Network and the multi-level reward.

We adopt the same training settings for all methods. For example, according to previous research [34], we set a maximum of 20 minutes for training and exploration on each app. We repeat the exploration process three times since different pieces of training may cause the model to follow different strategies [35]. To measure effectiveness, we perform descriptive statistics of the number of sensitive API calls discovered by the three methods, including mean, standard deviation, and median.

Figure 3 provides the quantitative results of the proposed approach and other baselines. We can observe that *GUIMind* outperforms *UniRLTest-S* and *UniRLTest*. Specifically, on average, *GUIMind* detects 2137 sensitive API calls compared to 1495 of the *UniRLTest-S* and 1442 of the *UniRLTest*. The result indicates that *GUIMind* can accelerate the empirical study by 43%, compared to *UniRLTest*. We also use the Mann-Whitney test to calculate the statistical significance and the Cliff's Delta (d) [36][2] to quantify the effect size of our experiments. We show the p-values and the effect size in Table II, which shows that *GUIMind* can meaningfully improve the speed of our empirical study.

We also conduct ablation studies on different types of inputs and investigate their impacts on model performance. Here, we eliminate the screenshot, the markup image, and the widget attributes, respectively. Overall, the performance of all three variants declines to various degrees. Specifically, compared to these three variants, *GUIMind* identifies 145, 325, and 470 more sensitive API calls, respectively. The results demonstrate that it is beneficial to improve the overall performance of *GUIMind* by integrating these types of inputs. This discrepancy suggests their significant effects on model performance ranked from low to high: the screenshot, the markup image, and the widget attributes.

[2]We follow accepted practical guidelines [35] for interpretation: large for $|d| \geq 0.474$, medium for $0.33 \leq |d| < 0.474$, small for $0.10 \leq |d| < 0.33$, and negligible for $|d| < 0.10$.

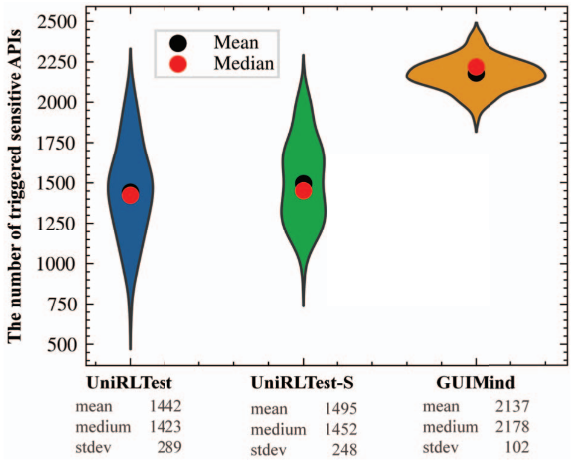


Fig. 3: Quantitative results of *GUIMind*, *UniRLTest-S*, and *UniRLTest* in discovering sensitive API calls.

TABLE II: Results of the Mann-Whitney test (adjusted p-value) and Cliff's Delta (d) when comparing the distributions of the number of sensitive APIs triggered.

Test	p-value	cliff-delta
GUIMind vs *UniRLTest-S*	< 0.001	0.28 (Small)
GUIMind vs *UniRLTest*	< 0.001	0.30 (Small)
UniRLTest-S vs *UniRLTest*	< 0.001	0.02 (Negligible)

V. Study Protocol

In this section, we discuss our experimental protocol for the measurement study. Specifically, we utilize *GUIMind* to detect such non-compliance behaviors. Our measurement study aims to answer the following research questions:

- **RQ1: What are the types of typical violations of the Data Minimization Principle?**
- **RQ2: How prevalent are privacy non-compliance issues among apps?**
- **RQ3: How do privacy non-compliance issues occur in different categories of apps?**
- **RQ4: Which types of personal data are most frequently leaked by apps?**
- **RQ5: What is the response from the administrators to the violations?**

A. Dataset

Our dataset includes 1876 realistic Android apps from the Xiaomi App Market, one of the largest app distribution platforms in China, between 2022-09-26 to 2022-10-09. Similar

TABLE III: The list of monitored sensitive APIs

Resource Types	Sensitive APIs	Descriptions
Telephony	APIs in TelephonyManager and SmsManager, e.g., getLine1Number(), getDeviceId(), sendTextMessage()	Accessing sensitive information and services related to telephony.
Location	APIs in Location, Address, LocationManagerService, and GoogleMap, e.g., getLastKnownLocation(), requestLocationUpdates(), getLongitude()	Retrieving device location data through direct GPS queries or by registering for GPS listeners.
Bluetooth	APIs in BluetoothDevice and BluetoothAdapter, e.g., getName(), getAddress(), getBondedDevices()	Accessing Bluetooth-related data and communications.
Microphone and Camera	APIs in AudioRecord, MediaRecorder, and Camera, e.g., startRecording(), start(), and takePicture()	Enabling the recording of audio and video, as well as capturing pictures.
Connectivity States	APIs in WifiManager and WifiServiceImpl, e.g., getScanResults(), getDhcpInfo(), and getWifiState()	Potentially exposing users' location or triggering harmful actions (based on the WIFI state changes).
Application States	APIs in PackageManager and UsageStatsManager, e.g., getInstalledPackages(), getApplicationInfo(), and queryUsageStats()	Analyzing the availability of installed antivirus and financial apps to identify malicious activities like phishing.
Accounts	APIs in AccountManager, e.g., getAccountsByType()	Accessing users' online accounts through a centralized registry.

to the dataset used in evaluating the performance of *GUIMind*, we adopted a systematic approach to fetch free apps based on the link (https://app.mi.com/download/{id}). The difference is that the IDs range from 1000 to 4000, rather than 1 to 1000. In other words, the apps used in our measurement study do not overlap with those used in evaluating *GUIMind*. We then randomly selected 2000 apps for our experiment. Finally, we removed the apps that crashed during the automatic interaction with Frida-Server. This left us with 1876 apps spanning 14 categories, which include health, sports, travel, etc. In addition, according to previous research [10], [37], [38] and malware analysis experience, the sensitive APIs monitored are shown in Table III.

B. Experiment Setup

In our experiments, we grant the system with all runtime permissions. Recall that we consider an app violates the Data Minimization Principle if it attempts to collect permissions beyond the users' expectations. Directly granting all permissions can avoid interrupting the exploration process without affecting our detection accuracy. Besides, the hardware for our experiments is the same as that for evaluating *GUIMind*.

C. RQ1: Types of Typical Violations

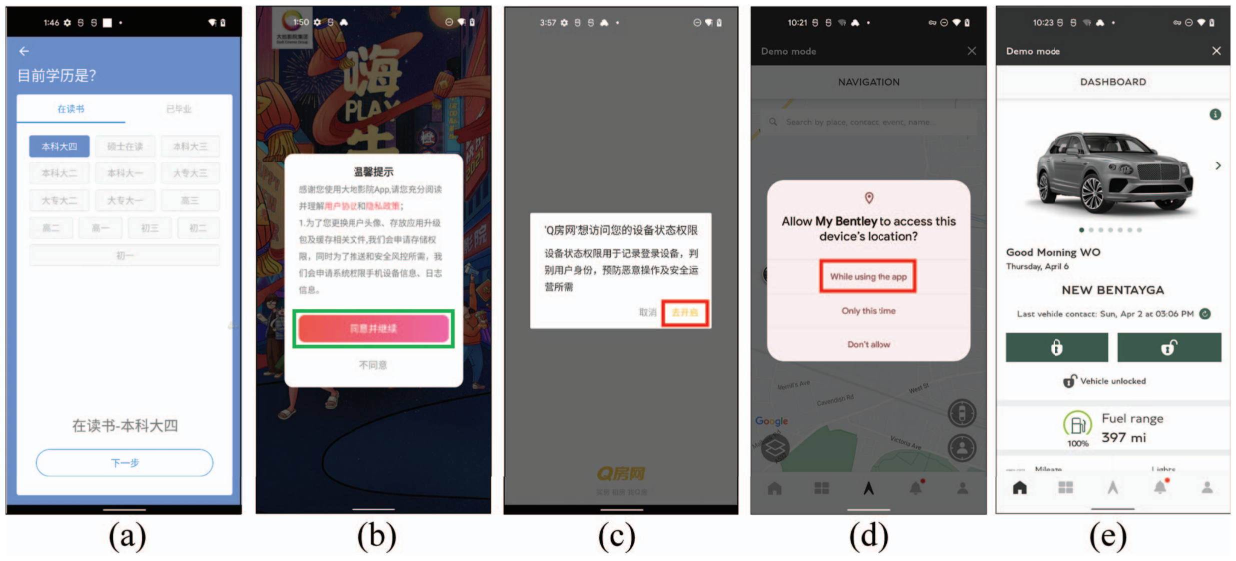
We notice four types of typical violations of the Data Minimization Principle. We discuss the details in this section.

Irrelevant Permissions. For 68.7% of violations, apps collect extra data beyond the functionalities they display to users. This practice violates GDPR Article 5(1)(b) [39], which requires that personal data must be collected for specified, explicit, and legitimate purposes and not further processed in a manner that is incompatible with those purposes. For example, Figure 4(a) shows the settings page for an educational experience. The app requests the device's IMSI, SIM card serial number, location information, and WIFI scan results, which are irrelevant to its intended purpose. Note that none of these permissions are protected by the runtime permission of Android. In practice, they are collected in stealth.

Bogus Agreements. The second most frequent non-compliance (16.9%) is bogus agreements. This indicates that apps may display an agreement that describes the desired permissions and appears to allow the user to choose whether to grant the requested permissions to the app. However, the choices of the users are not considered. Regardless of the users' choice, the app will collect the permissions it wants. This practice violates GDPR Article 6(1) [40], which requires that organizations must obtain consent from individuals before processing their personal data. For instance, Figure 4(b) is the privacy policy interface of a *Dadi Cinema* app. Before the user agrees to the privacy policy by clicking the green box, the app has collected all the sensitive data it wants.

Mismatched Permissions with the Agreement. For 10.1% of violations, apps may ask for extra permissions that are not shown in their user agreement. This practice violates GDPR Articles 7(2) [41] and 4(11) [42], which require that consent must be freely given, specific, informed, and unambiguous. For instance, when the app in Figure 4(c) requests permission to access device information, it also collects the location and Bluetooth device address, which are not included in the message shown to users, after the user clicks the "open" button marked by the red box.

Holding Permissions. Lastly, for 4.3% of violations, apps may access permission through a legitimate request in one activity and then abuse that permission in another to steal user data. This practice violates GDPR Article 5(1)(a) [39], which requires that personal data be processed lawfully, fairly, and transparently. For example, Figures 4(d) and Figure 4(e) show GUI screenshots of a *My Bentley* app. Figure 4(d)



感谢您使用大地影院App,请您充分阅读并理解用户协议和隐私政策;
1.为了您更换用户头像、存放应用升级包及缓存相关文件,我们会申请存储权限，同时为了推送和安全风控所需，我们会申请系统权限手机设备信息、日志信息。

'Q房网'想访问您的设备状态权限
设备状态权限用于记录登录设备，判别用户身份，预防恶意操作及安全运营所需

Fig. 4: Some concrete examples of privacy non-compliant apps.

requests location permission for a legitimate purpose, such as navigation services, and the user grants it by clicking the "While using the app" button marked by the red box in Figure 4(e). When the app reaches Figure 4(e), it abuses the granted permission to access location information beyond what is necessary. Holding the location allows the app to continuously track the movement of users, but the users do not expect such continuous tracking in this app.

Answer to RQ 1: Our study distills four types of typical violations of the Data Minimization Principle, which include irrelevant permissions (68.7%), bogus agreements (16.9%), mismatched permissions with the agreement (10.1%), and holding permissions (4.3%).

D. RQ 2: Prevalence of Violations

To answer this research question, we need to measure the privacy compliance of each subject app and then perform statistical analysis on all apps. In our measurement, we use the number of violations of the Data Minimization Principle in an app. Formally, we measure the number of violations by counting the activity-API pairs that break the Data Minimization Principle during the 20-minute testing. For instance, if an app collects a user's location information, contacts, and call history beyond what is necessary, there are three instances that violate the Data Minimization Principle. The higher the numerical value, the more severe the privacy non-compliance of the app. We define an app with 0 violations as fully compliant with privacy requirements.

Based on our research findings, a considerable number of mobile apps exhibit significant privacy non-compliance. After examining 1876 apps, we observe that 83.5% of them have at least one privacy non-compliance, while only 16.5% fully comply with all privacy requirements, as shown in Figure 5. These results indicate that most apps fail to follow best practices and principles when dealing with user data, which may expose users to potential risks and losses. As a result, developers and app distribution platforms must strengthen the review and oversight of their products and services.

In addition, we group apps with privacy non-compliance into three levels based on the number of violations: low (1-4), moderate (5-8), and high (9-12). Our analysis reveals significant disparities in privacy non-compliance across these groups, as shown in Figure 5. Specifically, out of all apps, 44.9%, 25.3%, and 13.3% belong to the first, second, and third categories, respectively. This suggests that most apps with privacy concerns only contain a limited number (1-4) of privacy non-compliance.

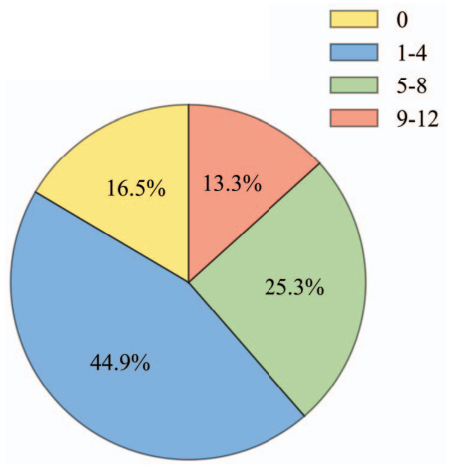


Fig. 5: Distribution of privacy non-compliance counts among apps.

Answer to RQ 2: Our study reveals that 83.5% of apps in our dataset contain at least one privacy violation, while only 16.5% fully comply with all privacy requirements. Thus, privacy non-compliance is prevalent in Android apps.

E. RQ3:Violations in Different Categories of Apps?

To address this research question, we break down privacy non-compliance into different categories, which include health, travel, news, etc. They reflect the various functions or services that the app provides to users. For example, in travel apps, users can use a range of services such as booking flights, hotels, and rental cars; in news apps, one can access and consume various types of news content such as articles, videos, podcasts, and live streams. Others are also commonly used in daily life. In our measurement, we use the average number of privacy non-compliant cases in each category of apps, namely the total number of privacy non-compliant instances divided by the number of apps in that category, as an indicator. Figure 6 shows the numerical results broken down by app categories.

From Figure 6, it can be seen that there are significant differences in the average number of privacy non-compliant issues among different categories of apps. Among the categories we examine, health, sports, and financial apps perform the worst regarding privacy non-compliance. They exceed others significantly with the average number, reaching 11.3, 9.5, and 9.1, respectively. The results reveal the impact and challenges of different functions or services on user privacy protection. This suggests that sports, health, and financial apps are the most in need of improvement and regulation. One possible reason is that these apps hold large amounts of sensitive data and are not careful enough when collecting the data, leading to privacy violations. Moreover, numerous commercial interests in these fields may motivate companies to prioritize business benefits over user privacy protection.

Furthermore, several categories with low privacy violations warrant special consideration. Figure 6 shows that three common app categories, including photography, productivity, and tools, exhibit low privacy non-compliance. On average, they only present 3.5, 2.5, and 3.2 privacy non-compliance issues, respectively (significantly lower than the overall level), with minimal variations (i.e., standard deviations of only 2.7, 2.3, and 1.8). A potential explanation is that these app categories have relatively simple functions and services and do not require too many permissions.

Answer to RQ 3: Our study reveals that app categories lead to diverse privacy violations. Among these categories, health, sports, and financial apps perform worst, with an average number of violations of 11.3, 9.5, and 9.1, respectively.

F. RQ4: Frequently Abused Permissions?

To answer this research question, we report the percentage of apps that illegally access personal data to all non-compliant apps. We distinguish non-compliant behaviors and categorize them into different types of personal data, such as locations, telephony, or Bluetooth. Finally, we utilize the statistical method to calculate the distribution of each type. This provides a more comprehensive picture of non-compliant access to personal data.

According to our statistical results, we observe that 71.1% of the apps access telephony information in a non-compliant manner, which is the most frequently leaked personal data type. This suggests these apps may obtain information such as device identifier (via *getDeviceId*), phone number (via *getLine1Number*), SIM serial number (via *getSimSerialNumber*), and IMEI number (via *getSimOperator*). This information can reveal users' identities, preferences, and habits and deliver personalized advertisements. The second largest non-compliance is accessing location data (25.8%), which is concerning given the potential impact of location tracking on privacy.

In addition to the above analysis, we discover a relatively high proportion (13.7%) of non-compliant access to Bluetooth information. This includes sensitive details such as bonded Bluetooth devices, names, and addresses. Such information can infer the user's identity, device type, and proximity to other devices or users. Further analysis can reveal other patterns and trends. Specifically, certain types of apps are more inclined to engage in non-compliant access to Bluetooth data than

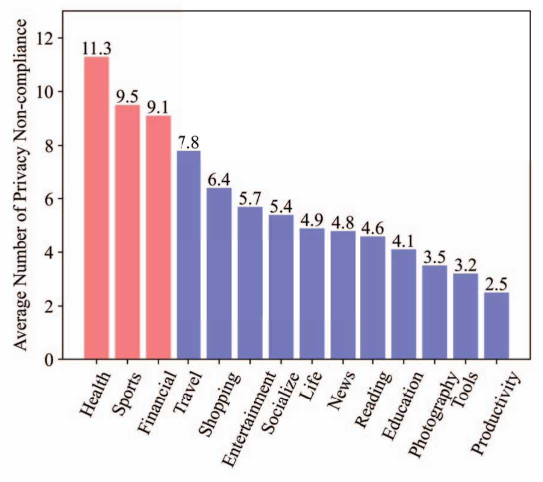


Fig. 6: Average number of privacy non-compliance by app category.

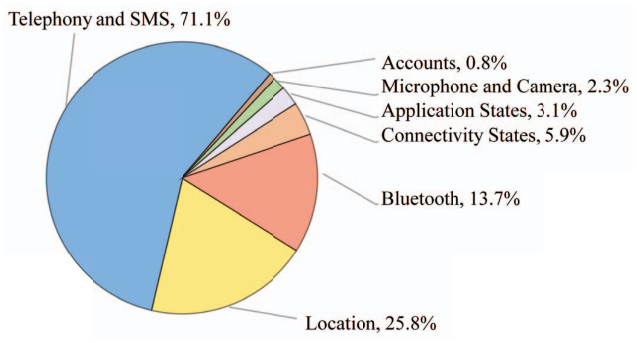


Fig. 7: Distribution of app privacy non-compliance by different personal data types.

other categories of apps. For example, a social app accesses Bluetooth information to find and connect with friends among nearby users. It is also possible to identify specific factors, such as the interests and preferences of app users, based on the devices they typically connect to.

Answer to RQ 4: Our study reveals that the most frequently leaked personal data type (71.1%) is telephony information, followed by location data (25.8%) and Bluetooth information (13.7%).

G. RQ5: Responses from the Administrator

To investigate the practicality of our tool, we randomly select 60 apps that violate the Data Minimization Principle at the activity level and submit them to the Personal Information Protection Task Force on Apps (PIPTFoA) for MANUAL inspection through their official website. The website provides a detailed report of the app's behavior and the reasons for the violation. The PIPTFoA is a credible institution jointly established by four authorities of China: the Ministry of Public Security, the Cyberspace Administration, the State Administration for Market Regulation, and the Ministry of Industry and Information Technology [43]. It is responsible for monitoring and evaluating the data collection practices of mobile apps. We only submit 60 apps because PIPTFoA has limited personnel, and we do not want to cause denial-of-service to the team.

Until the submission of this paper, 43 apps have been confirmed, of which eight are in severe violation and subsequently taken down from the app store. The team also agrees that the other 35 have mild violations. Besides the 43 apps, another ten apps have fixed their problems in the latest version. Overall, this result implies that our tool has considerable potential to identify apps that violate the Data Minimization Principle.

Answer to RQ 5: Our study indicates that out of 60 submissions, 43 apps have been confirmed, and another ten have fixed their problems in the latest version. This implies the practicality of our tool in detecting violations of the Data Minimization Principle.

H. Implications and Suggestions

This research demonstrates a pervasive presence (83.5% of apps) of violations of the Data Minimization Principle in Android apps, which could lead to private data leakage and other potential security risks. Moreover, our findings highlight several underlying factors and characteristics contributing to such non-compliance, such as typical violations, app categories, and types of personal data. This could offer valuable insights into developing more effective privacy protection measures.

For violations, we have the following suggestions. First, developers should confirm the necessary permissions for each activity before development. Second, automated testing tools, such as those proposed in our research, should be developed and implemented to identify and fix potential violations. Third, permission control should be designed at a finer-grained level, such as adjusting from the app to the activity level. The activity level indicates that each activity should only access the permissions it needs instead of the current permission mechanism, which gives access to all permissions of the apps. Finally, user awareness of app permission requests should be increased. For example, runtime permission prompts could be provided with detailed explanations, allowing users to understand better what the app requires.

VI. Discussion

A. Threats to Validity

Internal Threats. The main threat to internal validity is the hyperparameter setting, which can affect the performance of our tool. Due to time and resource constraints, we cannot perform comprehensive fine-tuning for hyperparameters. Therefore, the current parameters of our experiments may be sub-optimal. To mitigate this threat, we set all parameters to default or recommended values from previous studies. For cases without known references, we conduct small-scale experiments to select parameters based on best practices.

Another threat is using the same time budget for apps with various complexities. Our intention is not to provide an equal opportunity for all apps to explore their full potential but to establish a standardized performance metric within a specific time frame. This allows us to compare the efficiency and effectiveness of different tools in a controlled manner. We refer to previous research [34] for the particular time budget settings.

External Threats. Threats to external validity mainly result from the limited number of apps used in the experiments. To mitigate this threat, we adopt a systematic approach to construct the datasets, which retrieve apps from the Xiaomi app store (https://app.mi.com/download/{id}) based on an auto-increment ID allocating strategy. Although it is preferable to conduct evaluations on more apps, our analysis covers a substantial number of apps belonging to various classes, indicating the general suitability of our approach.

Downloading apps only from the Xiaomi app store may not give us a complete view of the Android app market. However, the Xiaomi app store is one of the largest and most popular app markets in the Android ecosystem. We believe that the apps available on the Xiaomi app store can reflect the general characteristics and trends of the Android app market. Moreover, narrowing our scope to one prominent app store allows us to conduct a more manageable study while providing valuable information that can be extended to other markets with further research. Besides, there may be a bias in manually labeling the consistency of $< activity, API >$ pairs. Nonetheless, previous work [44] indicates that senior students can serve as qualified agents in a well-controlled environment.

B. Limitations

One limitation of our empirical study could be the presence of statistical bias. Despite our efforts to select a diverse range of mobile apps for analysis, our sample size is still limited.

This may not represent the full heterogeneity of the population of mobile apps. Furthermore, our research findings regarding typical violations rely on manual inspection, which may be subject to individual bias and interpretation differences. Therefore, our results may not be applicable to other situations or contexts.

C. Future Work

Dynamic testing can be inefficient when handling large datasets such as AndroZoo, a common challenge for many dynamic analysis tools. However, there are some potential ways to improve the efficiency of *GUIMind* in the future, such as using parallel processing to run multiple instances of *GUIMind* on various machines or cloud platforms, applying sampling techniques to select a representative subset of APKs from the large dataset based on criteria such as popularity, diversity, or similarity, and adopting heuristic methods to prioritize the execution of APKs that are more prone to exhibit malicious behaviors or vulnerabilities. In addition, we can also explore other RL techniques besides DQN.

VII. Related Work

Privacy Violation Analysis. Different techniques have been proposed to detect privacy violations in mobile apps [45]–[59] and identify third-party tracking services through dynamic [10], [60]–[64] or static analysis [34], [65]–[71]. For example, Nguyen *et al.* [64] conducted a large-scale study on the consent notices of third-party tracking and investigated their violation of GDPR in Android apps. They first utilized image processing and natural language processing techniques to analyze consent user interfaces of apps and categorized them into four interaction mechanisms. Based on these mechanisms, an automated approach was proposed to detect personal data sent from apps. The results revealed that 20.54% of the apps utilized in their evaluation contained at least one violation of the GDPR's consent. Slaven *et al.* [48] proposed a semi-automated framework to check the consistency between privacy policies and their apps' code. In an empirical evaluation conducted on 477 Android apps, 341 potential privacy policy violations were identified using this framework.

Moreover, Pan *et al.* [31] proposed a flow-level system, FlowCog, to perform semantic extraction and inference for each information flow based on a Java-implemented taint-analysis tool [68]. However, it is a static analysis method, which is practically infeasible due to the intense usage of dynamic features such as code obfuscation, code encryption, dynamic class loading, and reflection. In contrast, APICOG [10] is a dynamic method that judges the legitimacy of sensitive API calls based on UI context, including text and image attributes.

GUI Testing. Early approaches used random strategies to automate mobile app testing, such as Monkey [17] and Dynodroid [72]. To improve code coverage, some researchers attempted to generate high-quality test cases using state machines (e.g., Stoat [73], Droidbot [33], and ORBIT [74]) or systematic strategies (e.g., SAPIENZ [75] and A^3E [76]). There are also some studies based on machine learning for automated GUI testing [77]–[83]. Currently, reinforcement learning techniques have been widely used in testing tasks [77], [78], [82]. For instance, Retecs *et al.* [84] applied reinforcement learning to regression testing, which drives the prioritization and selection of test cases more efficiently. Zhang *et al.* proposed a reinforcement learning model *UniRL-Test* [32] to thoroughly explore app states with a curiosity-driven strategy. However, they are general testing frameworks that maximize code or GUI coverage and cannot effectively trigger specific targets, such as sensitive API calls.

VIII. Concluding Remarks

In this paper, we present the first systematic study on violations of the Data Minimization Principle in mobile apps. This contribution is significant as prior research has yet to extensively explore mobile app privacy concerns, and our proposed approach provides a viable solution to address these issues. We first propose a new automated tool, *GUIMind*, to detect such violations in Android apps, which utilizes a deep reinforcement learning-based model to efficiently identify sensitive API calls to accelerate our empirical study. We then thoroughly examine instances of privacy non-compliance across various mobile apps and provide an in-depth analysis of the root causes behind these violations. Overall, our work provides valuable insights into the state-of-the-art techniques for detecting and mitigating privacy non-compliance in mobile apps and could serve as a foundation for future research in this field.

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] GDPR, "Complete guide to gdpr compliance," 2018. [Online]. Available: https://gdpr.eu/

[2] Brazil, "General personal data protection act (lgpd)," 2018. [Online]. Available: http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/L13709compilado.htm

[3] California, "California privacy rights act," 2018. [Online]. Available: https://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201720180AB375

[4] U. Kingdom, "Data protection act," 2018. [Online]. Available: https://www.legislation.gov.uk/ukpga/2018/12/contents/enacted

[5] China, "Personal information protection law," 2021. [Online]. Available: http://www.npc.gov.cn/npc/c30834/202108/a8c4e3672c74491a80b53a172bb753fe.shtml

[6] B. Shen, L. Wei, C. Xiang, Y. Wu, M. Shen, Y. Zhou, and X. Jin, "Can systems explain permissions better? understanding users' misperceptions under smartphone runtime permission model," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 751–768. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/shen-bingyu

[7] C. W. Munyendo, Y. Acar, and A. J. Aviv, ""desperate times call for desperate measures": User concerns with mobile loan apps in kenya," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2304–2319.

[8] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall, "A conundrum of permissions: installing applications on an android smartphone," in *Financial Cryptography and Data Security: FC 2012 Workshops, USEC and WECSR 2012, Kralendijk, Bonaire, March 2, 2012, Revised Selected Papers 16*. Springer, 2012, pp. 68–79.

[9] 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.

[10] 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.

[11] 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.

[12] P. Rajivan and L. J. Camp, "Influence of privacy attitude and privacy cue framing on android app choices." in *WPI@ SOUPS*, 2016.

[13] P. Wijesekera, J. Reardon, I. Reyes, L. Tsai, J.-W. Chen, N. Good, D. Wagner, K. Beznosov, and S. Egelman, "Contextualizing privacy decisions for better prediction (and protection)," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–13.

[14] 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*, 2012, pp. 1–14.

[15] R. A. Soni, "A study on android application development," *Journal of Telematics and Informatics*, vol. 1, no. 2, pp. 89–96, 2013.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

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

[26] OpenATX, "Uiautomator2," 2017. [Online]. Available: https://github.com/openatx/uiautomator2/releases

[27] R. Smith, "An overview of the tesseract ocr engine," in *Ninth international conference on document analysis and recognition (ICDAR 2007)*, vol. 2. IEEE, 2007, pp. 629–633.

[28] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques." in *NDSS*, 2016.

[29] Facebook, "Pytorch," 2016. [Online]. Available: https://github.com/pytorch/pytorch

[30] Oleavr, "Frida," 2016. [Online]. Available: https://github.com/frida/frida/releases

[31] X. Du, X. Pan, Y. Cao, B. He, G. Fang, Y. Chen, and D. Xu, "Flowcog: Context-aware semantic extraction and analysis of information flow leaks in android apps," *IEEE Transactions on Mobile Computing*, 2022.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[37] 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.

[38] 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.

[39] GDPR, "Art.5 principles relating to processing of personal data." 2021. [Online]. Available: https://gdpr.eu/article-5-how-to-process-personal-data/

[40] GDPR, "Art.6 lawfulness of processing." 2021. [Online]. Available: https://gdpr.eu/article-6-how-to-process-personal-data-legally/

[41] GDPR, "Art.7 conditions for consent." 2021. [Online]. Available: https://gdpr.eu/article-7-how-to-get-consent-to-collect-personal-data/

[42] GDPR, "Art.4 definitions." 2021. [Online]. Available: https://gdpr.eu/article-4-definitions/

[43] 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

[44] 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.

[45] N. Momen, M. Hatamian, and L. Fritsch, "Did app privacy improve after the gdpr?" *IEEE Security & Privacy*, vol. 17, no. 6, pp. 10–20, 2019.

[46] 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.

[47] S. Koch, M. Wessels, B. Altpeter, M. Olvermann, and M. Johns, "Keeping privacy labels honest," *Proceedings on Privacy Enhancing Technologies*, vol. 4, pp. 486–506, 2022.

[48] 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 *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 25–36.

[49] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?" in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 538–549.

[50] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. Bellovin, and J. Reidenberg, "Automated analysis of privacy requirements for mobile apps," in *2016 AAAI Fall Symposium Series* 2016.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5*. Springer, 2012, pp. 291–307.

[56] E. P. Papadopoulos, M. Diamantaris, P. Papadopoulos, T. Petsas, S. Ioannidis, and E. P. Markatos, "The long-standing privacy debate: Mobile websites vs mobile apps," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 153–162.

[57] M. H. Meng, Q. Zhang, G. Xia, Y. Zheng, Y. Zhang, G. Bai, Z. Liu, S. G. Teo, and J. S. Dong, "Post-gdpr threat hunting on android phones: dissecting os-level safeguards of user-unresettable identifiers," in *The Network and Distributed System Security Symposium (NDSS)*, 2023.

[58] D. S. Guamán, J. M. Del Alamo, and J. C. Caiza, "Gdpr compliance assessment for cross-border personal data transfers in android apps," *IEEE Access*, vol. 9, pp. 15 961–15 982, 2021.

[59] M. Fan, L. Yu, S. Chen, H. Zhou, X. Luo, S. Li, Y. Liu, J. Liu, and T. Liu, "An empirical evaluation of gdpr compliance violations in android mhealth apps," in *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*. IEEE, 2020, pp. 253–264.

[60] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 1021–1036.

[61] I. Reyes, P. Wijesekera, J. Reardon, A. Elazari Bar On, A. Razaghpanah, N. Vallina-Rodriguez, S. Egelman *et al.*, ""won't somebody think of the children?" examining coppa compliance at scale," in *The 18th Privacy Enhancing Technologies Symposium (PETS 2018)*, 2018.

[62] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 499–514.

[63] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1043–1054.

[64] T. T. Nguyen, M. Backes, and B. Stock, "Freely given consent? studying consent notice of third-party tracking and its violations of gdpr in android apps," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2369–2383.

[65] T. T. Nguyen, M. Backes, N. Marnau, and B. Stock, "Share first, ask later (or never?)-studying violations of gdpr's explicit consent in android apps," in *USENIX Security Symposium*, 2021.

[66] B. Andow, S. Y. Mahmud, J. Whitaker, W. Enck, B. Reaves, K. Singh, and S. Egelman, "Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with policheck," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[67] H. Wang, Y. Li, Y. Guo, Y. Agarwal, and J. I. Hong, "Understanding the purpose of permission use in mobile apps," *ACM Transactions on Information Systems (TOIS)*, vol. 35, no. 4, pp. 1–40, 2017.

[68] 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.

[69] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 217–228.

[70] 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 *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 578–592.

[71] K. Kollnig, R. Binns, P. Dewitte, M. Van Kleek, G. Wang, D. Omeiza, H. Webb, and N. Shadbolt, "A fait accompli? an empirical study into the absence of consent to third-party tracking in android apps," *arXiv preprint arXiv:2106.09407*, 2021.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.